An Approach To Component Generation and Technology Adaptation

James Randall Kipps

Dissertation

Technical Report No. 91-79

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

UNIVERSITY OF CALIFORNIA

IRVINE

An Approach To Component Generation and Technology Adaptation

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

James Randall Kipps

Dissertation Committee:

Professor Daniel D. Gajski, Chair

Professor Michael J. Pazzani

Professor Dennis F. Kibler

1992

The dissertation of James Randall Kipps is approved,

and is acceptable in quality and form for

publication on microfilm:

_____

_____

_____

Committee Chair

University of California, Irvine

1992

Dedication

This dissertation is dedicated to my lovely wife Donna Kipps,
without whose understanding and seemingly limitless
patience I could not have completed this effort,
and to our daughter Rachael Olivia Kipps, whose presence in our
lives places everything else into the proper perspective.

Contents

List of Figures

Acknowledgments

Having been a graduate student far too long, I have accumulated many people to acknowledge. First and foremost, I would like to thank my advisor Dan Gajski for taking me on late in the game and for giving me a problem and letting me run with it. Proof that the Gajski Plan works. I would also like to thank the other members of my doctoral committee, Mike Pazzani and Dennis Kibler.

Graduate school is like a river; long, seemingly endless but with a current that carries you on of its own. You can paddle hard and get downriver in two hours where you would get in three hours by drifting. I would like to acknowledge all the other paddlers and drifters I have know along the way and who have made the trip more rewarding. In particular, I would like to thank my friends Jim Wogulis, Dave Schulenberg, and Bernd Nordhausen, with whom I drifted down the Yukon. I would also like to acknowledge Randy Jones, Rogers Hall and Kären Wieckert, David and Mary Woo, Hadar Ziv, Mats Heimdahl, Chris Truxaw, David and Stephanie Aha, John Gennari, Wayne Iba, Dave Ruby, John Allen, Tim Cain, and Piew Datta, as well as Harry Yessayan, Craig Snider, Craig MacFarlane, Kari Forester, and Debi Brodbeck.

I would like to thank Ted Hadley for his hacking assistance with X and other software, and I would like to thank Nikil Dutt, Elke Rundensteiner and Allen Wu for insightful discussions that have contributed to this dissertation. Finally, I would like to thank Sanjiv Narayan, Frank Vahid, Loganath Ramachandran, Roger Ang, Viraphol Chaiyakul, and others in the UCI CADLab for their assistance and fellowship.

I would also like to acknowledge The RAND Corporation, which kept me employed during my too many years in graduate school, with special thanks to Jed Marti, Iris Kameny, Dave McArthur, Stephanie Cammarata, Steve Bankes, and others at RAND.

But most importantly, I would like to thank my wife Donna Kipps, my parents Harry and Barbara Kipps, my wife's parents Don and Effie Van Buskirk, and my entire family for their love and support throughout this long experience.

This work was supported in part by grants 91040 and 91041 from TRW and Rockwell International and by funding from contract NSF MIP-8922851.

Abstract of the Dissertation

An Approach To Component Generation and Technology Adaptation

by
James Randall Kipps
Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 1992
Professor Daniel D. Gajski, Chair

Component generation is the task of mapping the abstract functional specification of register-transfer (RT) components, such as decoders and multiplexers, adders and comparators, and multipliers and arithmetic logic units, into configurations of connected physical layout cells. Cells are drawn from a given ASIC (application-specific integrated circuit) library.

In this dissertation, I describe a symbolic pattern-matching approach to component generation and, relative to this, an approach to automating technology adaptation. I define the component decomposition algorithm and technology compilation algorithm that formalize these two approaches and describe implementations of each, in the DTAS component generation system and the LOLA technology adaptation system, respectively. I present empirical results to validate the utility of my approach to component generation, and I present a demonstration to validate my approach to technology adaptation.

My approach to component generation has two significant benefits. First, it enables the use of complex functional library cells, such as adders and CLAs, in the generation of designs for functional units. Second, it effectively searches the design space for designs that make desirable tradeoffs between design constraints, such as area and delay. My approach to technology adaptation is significant because it bootstraps the DTAS component generation system into new ASIC cell libraries, as well as cell libraries as they undergo change. In this way, the technology compilation algorithm automates the task of maintaining technology independence.

To validate my approach to component generation, I present the results of four sets of experiments using the DTAS component generation system. The first set examines the effectiveness of search control in DTAS; the second examines the capability to find desirable design alternatives; the third compares designs generated by DTAS with those of MISII; and the fourth shows how the use of complex library cells improves design quality. To validate my approach to automating technology adaptation, I demonstrate the application of the LOLA technology adaptation system to a cell library as it undergoes four phases of evolution.

Chapter 1
Design Synthesis and Technology Adaptation

1.1 Introduction

Synthesis means the combining of parts to form a whole. *Design synthesis* refers to the process of combining material parts to form the design of an artifact. The artifact is describe in the abstract by a behavioral or functional specification and by constraints on such aspects as its cost, performance, and manufacturability. The material parts represent the basic building blocks from which designs are constructed.

Heuristic techniques have been successfully applied to the design synthesis task in many domains. The appeal of such techniques is that they provide reliable solutions to search-intensive problems that are intractable by purely analytic means. After surveying systems for knowledge-based synthesis, Mittal and Frayman (1989) developed a precise definition of the general design synthesis task. One assumption of this definition is a "*fixed, pre-defined set of components* [i.e., material parts]...", an assumption they found to be common across the systems surveyed.

The work described here was sparked in part by the view that this "fixed parts" assumption is unrealistic. Further, relying on such an assumption severly limits the useful lifetime of a design tool. The material parts available to a design tool are dependent upon the fabrication technology used to produce those parts. As fabrication technologies change, so do the available parts. A design synthesis tool must be continually upgraded to keep pace with advances in fabrication technologies. Any tool that cannot be readily adapted to such technology changes faces eventual and rapid obsolescence.

I refer to the task of maintaining technology independence as *technology adaptation.* As the main topic of this dissertation, I present a knowledge-based approach

to a synthesis task in integrated circuit (IC) design referred to as *component generation.* The connection to technology adaptation is that component generation requires technology-specific design knowledge to achieve high levels of design quality.

1.2 Problem Definition

A component generation system maps the abstract functional specification of register-transfer (RT) components, such as decoders and multiplexers, adders and comparators, and multipliers and arithmetic logic units, into configurations of connected physical layout *cells*, which are the "material parts" of the domain. Cells are drawn from a given ASIC (application-specific integrated circuit) library.

Layout cells can be as simple as one- and two-level Boolean gates, which are common across many layout libraries, or complex functional units, such as multiplexers, comparators, and adders, the availability and functionality of which often varies across libraries. Functional cells often outperform functionally equivalent configurations of Boolean cells, but nonstandard support makes functional cells problematic to use in a technology-independent manner.

The approach to component generation that I present in this dissertation takes advantage of functional cells using library-specific design knowledge. In this way, my approach can generate higher-quality designs than purely technology-independent approaches that rely strictly on Boolean logic. To keep my approach robust to upgrades within a cell library and to differences between libraries, I also present an adjunct approach to automating technology adaptation with regard to component generation.

1.3 Objectives

The larger objective of the research described here is to address issues in design synthesis and technology adaptation. In particular, this research focuses on the two issues:

1. How can the dependency of a design synthesis tool on the fabrication technology be limited without also limiting design quality.
2. How can the level of effort to manually upgrade a design synthesis tool be reduced while maintaining the integerity of the tool against changes in the fabrication technology.

These issues are explored in the domain of IC design and in the context of a knowledge-based design tool (KBDT) for component generation.

The task to be performed by the KBDT is to generate technology-specific designs from technology-independent specifications. The research objectives in developing the KBDT were to achieve a high level of performance along the following dimensions:

- **Proficiency**. The KBDT is proficient if it can generate designs that are comparable in quality to those generated by an experienced human designer, given the same component specification and available material parts. Herein, the quality of a design is measured by the area and maximum delay characteristics of a design.
- **Completeness**. The KBDT is complete if it can generate designs for (or *hit*) any point in the space of possible component specifications. Herein, the space of component specifications is described by the GENUS Library of generic components (Dutt, 1988).
- **Coverage**. The coverage of the KBDT is a measure of its ability to take advantage of material parts supported by a given fabrication technology. Herein, material parts are considered to be the components available in an application-specific IC (ASIC) vendor's library, such as a CMOS standard cell or macrocell library.
- **Robustness**. The KBDT is robust if it can maintain the quality of its designs in the face of changes to the available material parts. Herein, changes are introduced by switching ASIC vendor's libraries or by adding or deleting parts from an ASIC vendor's library.
- **Reliability**. This KBDT is reliable if it generates designs in a timely manner using "reasonable" computing and human resources. Herein, timeliness is a soft constraint considered to mean design times of under one hour; reasonable computing and human resources mean a single designer on a Sun 4 workstation.

These five dimensions are fundamentally related to issues in technology adaptation. First, by its definition, robustness assumes that the available material parts *will* change and, thus, denies use of the fixed parts assumption in building the KBDT. Second, the coverage of the KBDT degrades as a direct result of changes to the set of available parts. Third, when key material parts have been replaced or are otherwise unavailable, then completeness also degrades. Fourth, as coverage and completeness degrade, the proficiency of the KBDT, which is the ultimate goal, will likewise degrade. Reliability is important because it excludes approaches to design synthesis and technology adaptation that are computationally intractable.

There is one other noteworthy dimension to the KBDT along which high performance is a desired, but not major, research objective:

- **Correctness.** This measures the degree to which the functionality of generated designs can be validated.

Ideally, one would like to validate the knowledge within the design tool. For the KBDT, correctness will be verified by analysis and simulation of generated designs.

1.4 Approach

The KBDT for component generation is designed using a symbolic pattern-matching approach with a branch-and-bound search strategy. I formally define this approach with the *component decomposition algorithm.* I have implemented the component decomposition algorithm in the DTAS component generation system. To maintain technology independence, I augment the component decomposition algorithm with the *technology compilation algorithm.* The technology compilation algorithm uses a similar pattern-matching approach. I have implemented the technology compilation algorithm in the LOLA technology adaptation system.

DTAS and LOLA are related as shown in Figure 1.1. DTAS produces component designs from layout cells using a symbolic encoding of digital design knowledge. Much of this design knowledge is generic in the sense that it is applicable to most ASIC libraries. In the absence of library-specific design knowledge, DTAS is capable of generating complete designs down to the level of generic Boolean gates. To implement designs with cells from a given library, DTAS must be supplied with library-specific design knowledge. This knowledge can be as simple as mapping generic Boolean gates into available library gates, or as sophisticated as decomposing n-bit functional units into configurations of functional library cells. LOLA assists in the generation of library-specific design knowledge. Given the specification of the cells in a layout library, LOLA uses a symbolic encoding of design principles to generate design knowledge that is specific to the available library cells.

1.4.1 The DTAS Component Generation System

For its input, DTAS is supplied the functional specification of a netlist of generic RT components drawn from the GENUS library. DTAS outputs a set of alternative designs for the input netlist. Each alternative is represented as a hierarchical netlist that traces the top-down decomposition of the input netlist into subcomponents.

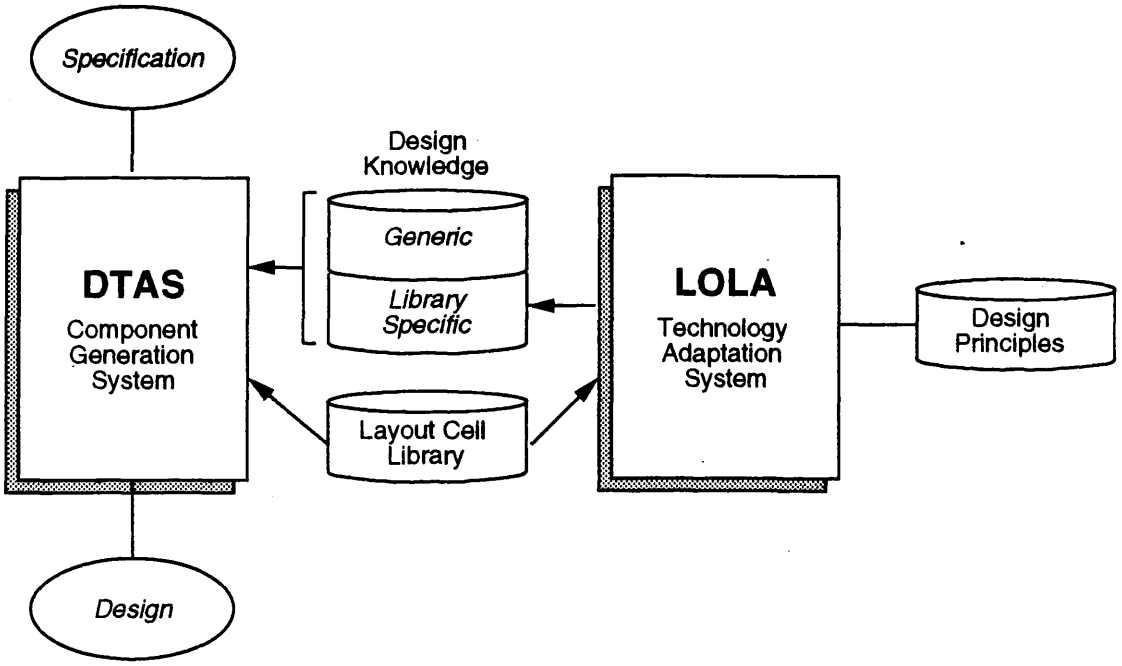


Figure 1.1: Relation of DTAS and LOLA.

The leaves of each netlist depict the configuration and connection of layout cells implementing the design.

Design knowledge is represented in DTAS with *decomposition methods*. Each method pairs a component specification pattern to a procedure for configuring a structural netlist. A method is *applicable* to any component specification that matches its specification pattern. The matching process binds variables in the specification pattern to corresponding literals from the component specification. The configuration procedure generates a netlist of connected subcomponents based on the values bound to the pattern variables. The connected subcomponents have specifications that can be matched by other methods and implemented by subsequent method expansion. The DTAS design process repeatedly expands applicable methods until all components have been mapped into layout cells.

The DTAS design process, as defined by the component decomposition algorithm, performs a branch-and-bound search through a space of design alternatives. Each decomposition method encapsulates a *design style* used in implementing a particular class of components. It is possible for more than one design style to exist for any class of components, most notably arithmetic components, and so it is possible for more than one decomposition method to be applicable to the same component

specification. DTAS explores the space of design alternatives by expanding every applicable method.

The size of the design space is constrained with a user-defined performance filtering function. When given a list of alternative designs for a subcomponent in the design hierarchy, the filtering function is expected to compare alternatives and discard those that do not meet desired performance characteristics. Possible filtering functions include discarding all but the smallest or fastest designs or all but some percentage thereof. The filtering function used in validating DTAS' performance discards all alternatives that do not make favorable trade-offs between area and delay. Using the filtering function as a way of bounding search, it is possible to find a small range of desirable designs from a space several hundred thousand potential alternatives.

1.4.2 The LOLA Technology Adaptation System

Technology dependencies are localized to material parts drawn from the cells in an ASIC library. Changes in the fabrication technology are introduced by upgrading the cells available in the library or by switching libraries. The effects of technology changes on DTAS are minimized with knowledge acquisition techniques for acquiring library-specific decomposition methods, also called *construction methods*. The acquired methods allow DTAS to take immediate advantage of new library cells. Many library-specific methods can be acquired automatically through the LOLA technology-adaptation system.

LOLA operates as a preprocess to synthesis. LOLA is invoked when DTAS is introduced to a new layout library or after changes appear in a familiar library. The inputs to LOLA include the cells available in the library and design principles indicating how to use the anticipated classes of library cells. LOLA applies these principles to generate decomposition methods that are specific to the library cells available. For example, when presented with a 4-bit adder cell, LOLA generates a method that decomposes n-bit adders into $\frac{n}{4}$ 4-bit adders.

Design principles are represented with *acquisition templates*. Acquisition templates pair two sets of component specification patterns with a procedure for defining decomposition methods. One set of specification patterns is called the *include* set; the other is called the *exclude* set. An acquisition template is applicable to any distinct set of library cells whose specifications completely match the specification patterns in the include set, given that there is not an additional set of library cells whose specifications completely match the specification patterns in the exclude set.

When an applicable acquisition template is expanded, it generates one or more construction methods. Like decomposition methods, each construction method pairs a component specification pattern to a procedure for configuring a netlist of connected subcomponents. A construction method differs from a decomposition method in that certain subcomponents in the configuration procedure are marked as instances of the library cells matched in the include set. For example, a method generated for the 4-bit adder cell from above would not only decompose n-bit adders into $\frac{n}{4}$ 4-bit adders but would identify the adder cell by name.

As defined by the technology compilation algorithm, LOLA performs an exhaustive search, expanding all applicable acquisition templates. There are plans to include two addition phases to LOLA: one that evaluates the performance of DTAS given the generated methods; and another that constrains methods that do not lead to desirable designs. However, these two phases have not been implemented.

1.5 Validation

In this dissertation, I make three claims. First, I claim that a symbolic pattern-matching approach to component generation, as defined by the component decomposition algorithm, can take advantage of library-specific design knowledge and complex functional library cells. Second, I claim that this approach encapsulates knowledge of design styles that can be used to explore the space of design alternatives and to find designs that make desirable trade-offs between design characteristics, in particular, area and delay. Third, I claim that a similar approach can be used to partially automate technology adaptation with regard to the component decomposition algorithm.

To validate the first two claims, I have run three sets of experiments using the DTAS component generation system. The first set of experiments shows how the search control principles of the component decomposition algorithm allow DTAS to find desirable designs from design spaces that are computationally intractable to enumerate. The second set shows how the encapsulation of design styles in decomposition methods allows DTAS to compare design alternatives and find ranges of designs that make desirable trade-offs between area and delay. The third set shows how the use of functional decomposition and functional specification allow DTAS to map designs into libraries of complex functional cells and generate higher-performance designs than it is possible to generate when mapping designs into a library of simple Boolean cells.

I validate my third claim, i.e., concerning my approach to technology adaptation, by demonstration. I show how acquisition templates can be used to generate library-specific decomposition methods for a cell library as it evolves through four library upgrades. With each upgrade, I run DTAS to show how the methods generated effect

design quality. For these experiments, I use DTAS to design a 64-bit 16-function arithmetic logic unit.

1.6 Contributions

With the dissertation research described here, I claim to make the following contributions:

1. **I describe and demonstrate an approach to component generation that takes advantage of complex functional library cells.**

   Current approaches use procedural module generators that either map designs to Boolean cells or do placement and routing. As a result, such approaches cannot take advantage of complex library cells that can improved design quality.

2. **My approach to component generation provides a formalism for encapsulating and applying alternative design styles, for exploring the space of design alternatives, and for finding designs that make favorable trade-offs.**

   Current approaches do little or no search between alternative design styles. A design style is selected on the basis of design constraints. Current approaches do not examine alternative design styles or consider that desirable trade-offs may be possible.

3. **I demonstrate that technology-specific synthesis is feasible when technology adaptation can be automated.**

   The desire to maintain technology independence has restricted previous approaches to component generation to using Boolean logic and graph-matching approaches, which are incapable of taking advantage of complex functional library cells.

1.7 Summary

The remainder of this dissertation is organized into eight chapters, a summary of which is given below.

- **Chapter 2: BACKGROUND AND RELATED RESEARCH**

  Overviews and relates the topics of IC design synthesis, knowledge-based design, and knowledge acquisition. References and surveys pertinent research contributions in each topic area.

- **Chapter 3: APPROACH**

  Describes the rationale behind my approach to component generation and technology adaptation. Motivates use of functional library cells and design space search. Explains the use of design principles in technology adaptation.

- **Chapter 4: THE COMPONENT DECOMPOSITION ALGORITHM**

  Formally defines the component decomposition algorithm, its data structures, and operations.

- **Chapter 5: COMPONENT DECOMPOSITION EXAMPLES**

  Steps through two applications of the component decomposition algorithm to further clarify concepts discussed in Chapter 5.

- **Chapter 6: THE DTAS COMPONENT GENERATION SYSTEM**

  Overviews the structure, design, and operation of the DTAS system, which implements the component decomposition algorithm.

- **Chapter 7: VALIDATING COMPONENT DECOMPOSITION**

  Describes the experiments run with DTAS to evaluate the effectiveness of the component decomposition algorithm. Presents summary of results.

- **Chapter 8: TECHNOLOGY COMPILATION AND LOLA**

  Formally defines the technology compilation algorithm and describes its relationship to the component decomposition algorithm. Overviews the structure and operation of the LOLA technology-adaptation system.

- **Chapter 9: VALIDATING TECHNOLOGY COMPILATION**

  Presents a demonstration of LOLA to a layout cell library as it evolves over time.

- **Chapter 10: CONCLUSIONS**

  Summarizes the results and contributions of this dissertation. Describes the future directions that this research can follow.

Chapter 2
Background and Related Research

In this chapter, I review topics in IC design synthesis as they relate to component generation. First, I overview approaches to component generation found in silicon compilation using logic synthesis; then, I present technology independence as one of the major issues in IC design synthesis; finally, I review knowledge-based design and knowledge acquisition, noting that technology independence has been overlooked as an issue in these areas. Pertinent research in these topic areas is cited and related to DTAS and LOLA.

2.1 IC Design Synthesis

The task of automating the design synthesis process for digital logic and integrated circuits (ICs) has been a primary objective of the computer-aided design (CAD) community for several decades. In this domain, the artifact to be synthesized is an IC system; the material parts from which it is synthesized are (ultimately) the transistors printed on silcon. The current trend is to factor IC design into layered subtasks, where the material parts of one level become the abstract specification for the next.

Silicon compilation is an emerging technology based on this view of layered synthesis subsystems (Gajski and Thomas, 1988). At a high level, silicon compilation is a three-tiered process of *behavioral synthesis*, *logic synthesis*, and *layout synthesis*. Behavioral synthesis maps system behavior into structure; logic synthesis maps generic structure to a technology base and optimizes it; and layout synthesis maps structure to silicon.

Component generation plays an intermediate role between behavioral and logic synthesis. The task of component generation is to map specifications of register-transfer (RT) components that appear in the output of behavioral synthesis into

technology-specific configurations of layout cells. The component decomposition algorithm defines a knowledge-based approach to component generation. This algorithm, as implemented in the DTAS component generation system, operates using technology-specific design knowledge. Technology independence is maintained with the technology compilation algorithm, as implemented in the LOLA technology adaptation system. LOLA can be viewed as a knowledge acquisition subsystem that upgrades DTAS's design knowledge as changes occur in its target cell library.

2.2 Silicon Compilation and Logic Synthesis

The term "silicon compilation" is attributed to Dave Johannsen (1979), who used it to describe Bristle Blocks, a system that synthesized large-scale integrated (LSI) circuits by assembling parameterized cells into a fixed-architecture layout. Since then, the term has appeared in a variety of contexts. In this dissertation, "silicon compilation" is used to refer to a software system or collection of loosely integrated systems that support IC design synthesis from a behavioral system description to layout. Examples of such silicon compilers include CMUDA (Director et al., 1981), MIMOLA (Marwedel, 1984), Cathedral II (de Man et al., 1986), Genesil (Cheng and Mazor, 1988), and the Yorktown Silicon Compiler (Brayton et al., 1988).

The high-level structure of a generic silicon compiler is depicted in Figure 2.1. In a design methodology based on synthesis, the designer begins by describing the behavior of a system using a hardware description language, such as VHDL. This level of description indicates the intended behavior and functionality of the system rather than its implementation. Behavioral synthesis focuses on data-flow/control-flow graph manipulations with the objective of optimizing register and operator scheduling and allocation (Thomas, 1986; Park and Parker, 1989; McFarland, Parker, and Camposano, 1990). The output of behavioral synthesis is typically a netlist of RT functional components, plus a state table describing control logic, plus other random combination logic.

Logic synthesis focuses on technology mapping and optimization of combinational logic. The output of logic synthesis is an optimized Boolean description mapped to layout cells, which is then passed through a phase of layout synthesis for placement and routing in silicon. Ideally, logic synthesis should start with the complete output of behavioral synthesis. However, the focus of current research in logic synthesis is on multilevel Boolean logic (Brayton and E. Detjens, 1986; Detjens et al., 1987; Keutzer, 1987; Brayton, Hachtel, and Sangiovanni-Vincentelli, 1990), which accounts for only about 20 percent of the high-level structural design, namely the sequencing and random logic (de Geus, 1989).

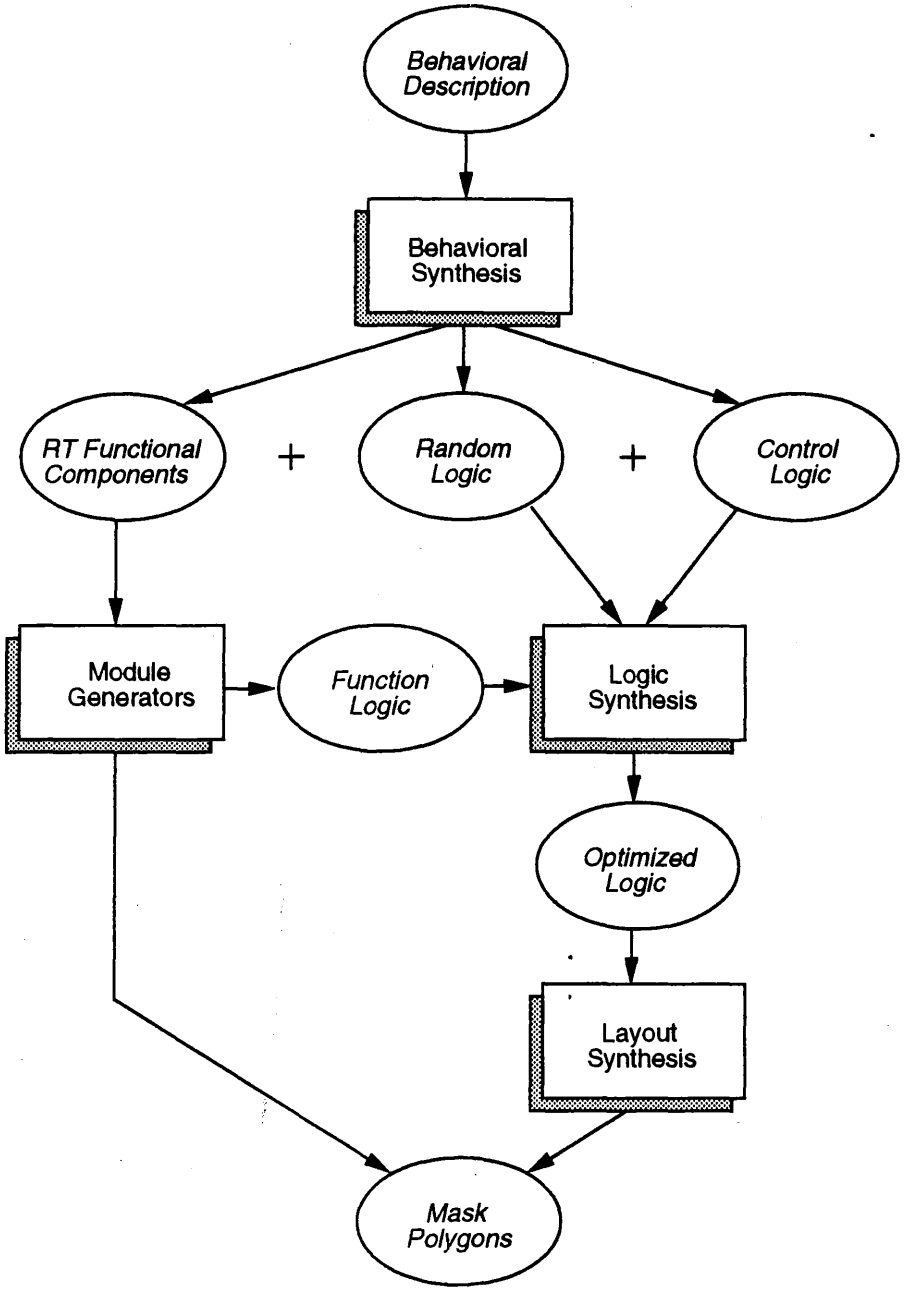


Figure 2.1: Structure of a silicon compiler.

The RT functional components are often mapped to layout with module generators. Module generators take the form of subroutines whose execution assembles an RT component from a library of small- (SSI) or medium-scale integrated (MSI) cells with a predefined layout. For instance, the Genesil silicon compiler bases its module generators, called block compilers, on a standard-cell methodology. Genesil provides block compilers for such elements as RAM, ROM, PLA, as well as ALUs, registers, and other data path elements; blocks are composed of predefined layout tiles that are assembled into arrays according to a tiling algorithm. Likewise, CATHEDRAL II, developed for high-complexity/low-performance digital signal processing applications, uses "expertly" crafted module-generator programs to produce layouts for functional building blocks, such as adders, registers, and shifters.

Some silicon compilers integrate RT synthesis with logic synthesis (Camposano and Trevillyan, 1989), using module generators that decompose RT functions into Boolean logic. In the Yorktown Silicon Compiler, memoryless combinational operators are mapped to modules written in the APL-like YLL language. YLL modules are expanded into Boolean networks that are passed through a phase of logic synthesis using algorithms from ESPRESSO-II (Brayton et al., 1984). Similarly, ICBD (Chen and Gajski, 1990) uses component generators to produce logic equations for functional components and then passes these equations plus performance constraints to the MILO logic optimizer (Vander Zanden and Gajski, 1988).

As illustrated in Figure 2.2, the DTAS component generation system subsumes the role of module generators in silicon compilation. One of the problems with module generators is that they represent a "black box" view of RT synthesis that does not preserve the hierarchical structure of component designs. Module generators output locally optimal configurations of simple cells that might not be globally optimal; without the hierarchical structure, redesign to achieve global optimality, i.e., switching from a ripple-carry adder to a carry look-ahead adder, is computation intractable. DTAS, on the other hand, represents a "declarative" view in which RT components are synthesized in discrete levels using decomposition methods. Each method describes how a particular (parameterized) class of components can be implemented by a configuration of subcomponents, which can be further decomposed by other methods. In this way, all levels of the design hierarchy are open to inspection by other tools, such as those for optimization and performance evaluation. Several methods can be written for decomposing the same class of component; each method depicts an alternative design style and permits an extensive exploration of the design space.

DTAS is also unique in its ability to map RT components to complex functional cells available in an ASIC library. Module generators cannot interface to arbitrary ASIC libraries (except at the level of SSI cells) without being reprogrammed. In DTAS, designs are grounded with cells from a given library, either vendor supplied

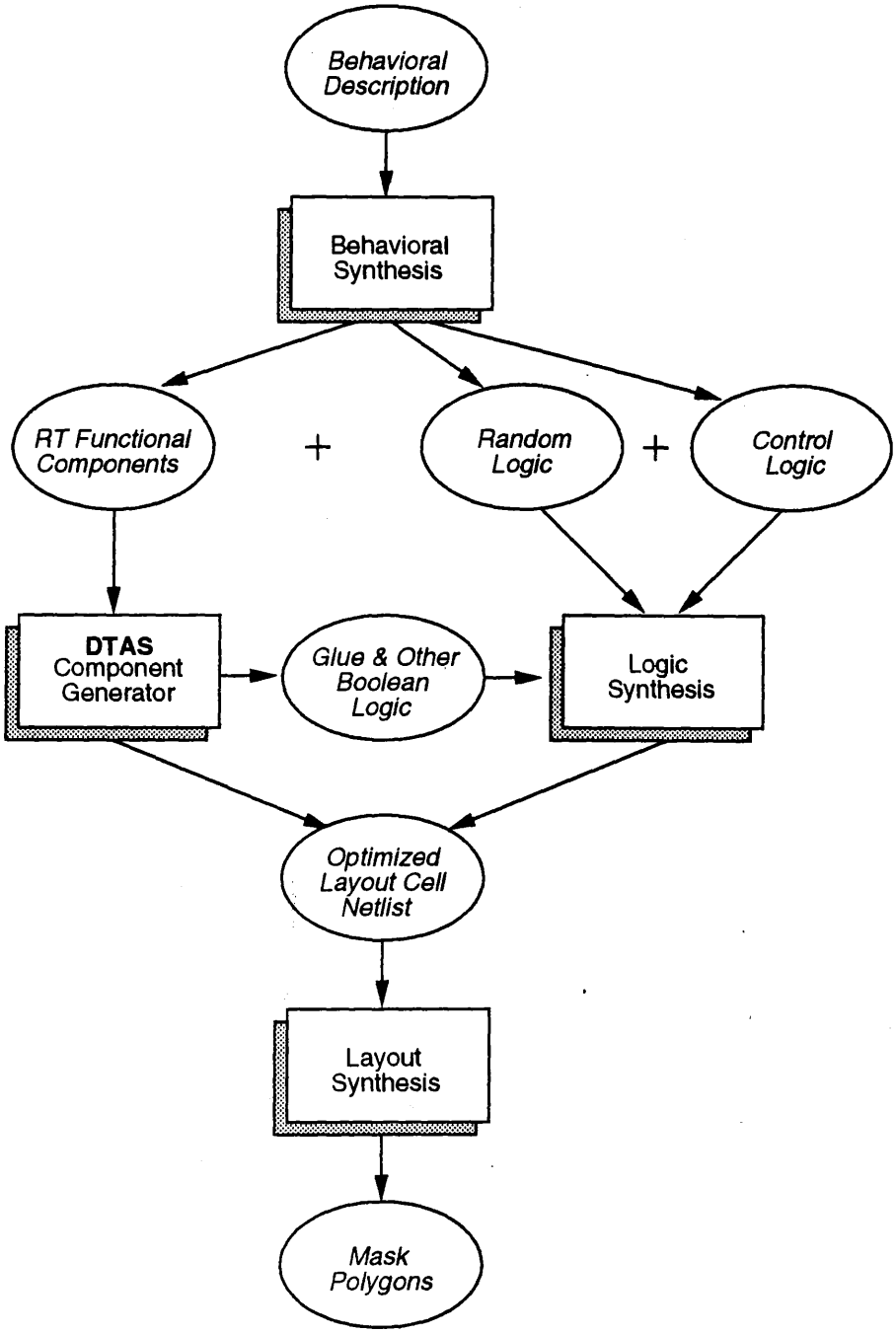


Figure 2.2: Role of DTAS in silicon compilation.

or custom. Decomposition methods can guide the design process towards implementations using MSI- and LSI-level cells. The portions of the design that use Boolean "glue" logic can be passed through logic synthesis for logic-level optimizations and layout. The portions of the design that use functional library cells can be passed through the vendor's layout tool.

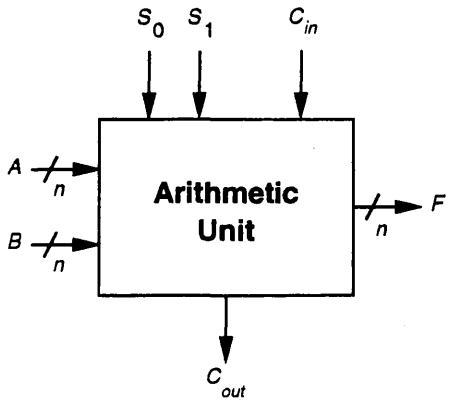
2.3 Technology Independence

One of the major issues in IC design synthesis is that of maintaining *technology independence* in technology mapping. For instance, if a design tool can map an IC system into a CMOS implementation can it also map this system into a Gallium-Arsenide implementation? The technology used in fabricating an IC system, as well as the foundry that manufactures the chips, dictates the physical details, such as fan-in, load, placement, and routing, that must be considered in order to generate high-quality designs. There are many fabrication technologies and foundries available today for manufacturing ICs. The characteristics of these technologies change often; they can be expected to continue changing well into the next century.

Early IC synthesis tools (Friedman and Yang, 1969; Johannsen, 1979; Director et al., 1981; Southard, 1983) demonstrated the potential for automation but used a "one-size-fits-all" fixed-architecture approach to technology independence. The weakness of this approach is its inability to take advantage of the technology-specific knowledge that human designers use to generate high-quality designs. As a result, the synthesized designs were too large and slow to be commercially viable.

The current approach to achieving technology independence, with regard to logic synthesis tools and module generators, is to localize dependence within a library of simple and commonly used IC cells, such as one- and two-level Boolean gates, and then synthesize designs from these cells (Brayton and E. Detjens, 1986; de Geus and Gregory, 1986; Keutzer, 1987; Bergamaschi, 1988). Maintaining such libraries against technologies changes is a relatively simple task. The weakness of this approach is that it requires designs to be decomposed to a very fine grain, i.e., the level of Boolean gates. Although all logic components can be described with Boolean equations, it is not always desirable to do so. As the number of inputs and outputs grows, Boolean descriptions become increasingly complex. The upper bound on the number of minterms in a Boolean equation grows exponentially with each new input. While each output requires only one Boolean equation, the complexity of factoring a set of equations is also bounded by the number of minterms that must be considered.

To emphasize the level of complexity involved with this approach, consider synthesizing the RT component shown in Figure 2.3. This component can perform four



S_0	S_1	F
0	0	A ADD B
0	1	A SUB B
1	0	INCR A
1	1	DECR A

Figure 2.3: Sample RT component: n-bit arithmetic unit.

arithmetic operations on two n-bit inputs A and B: addition, subtraction, increment A, and decrement A. The n-bit output of the operations is generated at F. S is a 2-bit function select line; C_{in} is an input carry; and C_{out} is an output carry. In the case of $n = 2$, there are 7 inputs and 3 outputs, requiring three Boolean equations with a possible 127 minterms each. When $n = 4$, there are 11 inputs and 5 outputs, requiring five equations with a possible 2048 minterms each. It is not too unrealistic to assume that existing logic synthesis tools can handle either of these cases. On the other hand, when $n = 64$, which is a realistic figure for today's processors, there are 131 inputs and 65 outputs, requiring 65 equations with a possible 2^{131} minterms each. While a Boolean description of this size can be computed, it cannot be minimized and factored by existing synthesis tools without the availability of considerable computing resources and without incurring considerable delay.

DTAS also localizes technology dependence in a set of library cells. The difference is in the complexity of the cells used. Library cells provide optimized layouts for commonly occurring logic circuits. Library cells are available at a variety of levels, from simple cells at the SSI level to complex cells at the MSI to LSI level. Simple cells include one-and two-level Boolean gates, 2-bit multiplexers, and 1- and 2-bit full adders. Complex cells include $>$ 2-bit decoders and multiplexers, $>$ 2-bit adders and comparators, as well as multipliers and ALUs, shift registers and counters. Complex cells are intended to provide better performance (e.g., smaller, faster, more powerful) than functionally equivalent configurations of simple cells. By using complex library cells, DTAS can obtain improved design quality and decrease the complexity of design synthesis.

One difficulty apparent in using complex cells stems from the fact that there is no uniform support for the same functions or sizes of complex cells across ASIC

vendor's libraries. For instance, one library might support a four function 16-bit ALU while another might support a 16 function 4-bit ALU. Nonuniform support makes movement between libraries difficult and degrades technology independence. DTAS overcomes this problem with the LOLA knowledge acquisition system. The purpose of LOLA is to acquire decomposition methods that are specific to the cells in a given library, both simple and complex. Thus, when moving between libraries, DTAS can discard its old library-specific methods for new methods generated by LOLA and, thus, maintain its technology independence while still accessing the complex cells of the library.

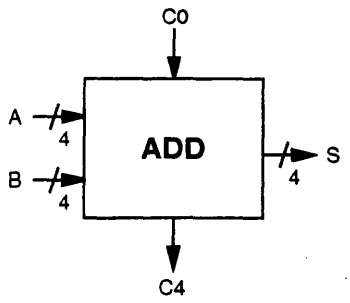
2.4 Mapping to Complex Functional Cells

Technology mapping to complex functional cells is becoming an important issue in IC design synthesis. As mentioned above, complex cells are provided by ASIC vendors as optimized layout of commonly occurring RT functional units, such as n-bit adders and multipliers. Complex cells provide improved performance over functionally equivalent configurations of Boolean cells, so their use in a design can improve overall design quality.

The primary reason that complex functional cells cannot be used by existing approaches to technology mapping in logic synthesis or in component generation is related to the computational complexity of subgraph matching and graph isomorphism. These approaches represent layout cells in a canonical Boolean form that is matched against a directed-acyclic graph representing the Boolean description of the components being synthesized (Keutzer, 1987; Detjens et al., 1987). The complexity of graph matching is not a significant factor when using simple subgraphs, so this technique works well for libraries of Boolean cells. As the number of inputs and outputs increases, however, the complexity of graph matching quickly becomes problematic, degrading the performance of a technology mapper by orders of magnitude and making the task computationally intractable.

DTAS overcomes the problem of mapping to complex functional cells by replacing the Boolean representation with an abstract symbolic representation, referred to as a *functional specification.* A functional specification represents library cells in terms of their function, i.e., AND, OR, ADD, MULT, and by their interface points, i.e., i/o ports and pins.

As an example, consider the 4-bit adder depicted in Figure 2.4(a). A functional specification for this adder is shown in Figure 2.4(a). This specification denotes that the function of the cell is `ADD`, that the cell has two 4-bit input ports, `A` and `B`, and a 1-bit input port `C0`, as well as a 4-bit output port `S` and a 1-bit output port `C4`.



(a)

<ADD,<[<A,4>,<B,4>,<C0,1>],[<S,4>,<C4,1>]>

(b)

S0 = C0'*(A0'*B0+A0*B0')+C0*(A0+B0')*(A0'+B0)

S1 = (A0'+B0')*(C0'+(A0+B0')*(A0'+B0))*(A1'*B1+A1*B1')+(A0*B0+C0*(A0'*B0+A0*B0'))*(A1+B1')*(A1'+B1)

S2 = (A1'+B1')*(C0'+(A0+B0')*(A0'+B0)+(A1+B1')*(A1'+B1))*(A0'+B0'+(A1+B1')*(A1'+B1))*(A2'*B2+A2*B2')
+(A1*B1+C0*(A0'*B0+A0*B0')*(A1'*B1+A1*B1')+A0*B0*(A1'*B1+A1*B1'))*(A2+B2')*(A2'+B2)

S3 = (A2'+B2')*(C0'+(A0+B0')*(A0'+B0)+(A1+B1')*(A1'+B1)+(A2+B2')*(A2'+B2))*(A0'+B0'+(A1+B1')*(A1'+B1)
+(A2+B2')*(A2'+B2))*(A1'+B1'+(A2+B2')*(A2'+B2))*(A3'*B3+A3*B3')
+(A2*B2+C0*(A0'*B0+A0*B0')*(A1'*B1+A1*B1')*(A2'*B2+A2*B2')
+A0*B0*(A1'*B1+A1*B1')*(A2'*B2+A2*B2')
+A1*B1*(A2'*B2+A2*B2'))*(A3+B3')*(A3'+B3)

C4 = A3*B3+C0*(A0'*B0+A0*B0')*(A1'*B1+A1*B1')*(A2'*B2+A2*B2')*(A3'*B3+A3*B3')
+A0*B0*(A1'*B1+A1*B1')*(A2'*B2+A2*B2')*(A3'*B3+A3*B3')
+A1*B1*(A2'*B2+A2*B2')*(A3'*B3+A3*B3')+A2*B2*(A3'*B3+A3*B3')

(c)

Figure 2.4: Example representation of a 4-bit adder cell: (a) graphical depiction; (b) functional specification; and (c) Boolean description.

This specification can be compared to the Boolean description specification shown in Figure 2.4(c).

Boolean logic can be viewed as a symbol-system language. It has a well-defined, widely-accepted semantics and, as a result, has a natural appeal for modeling cells and components. The abstract functional specification used by DTAS and illustrated in Figure 2.4(b) can also be viewed as part of a symbol-system language; not the language of Boolean logic but a different language. This symbol-system language can be given a semantics, and it can be used to model cells and components.

The appeal this functional specification language over the language of Boolean logic should be clear from Figure 2.4; namely, simplicity of representation. Using functional specification, a 4-bit adder cell can be represented as succinctly as a 2-input NAND gate; a 16-bit adder as succinctly as a 4-bit adder; a 64-bit ALU as succinctly as a 16-bit adder.

DTAS represents RT components using the same functional specification language that it uses to represent layout cells. Components are decomposed using a pattern-matching language based on this same functional specification representation. As a result, the complexity of technology mapping is that of matching function to function, ports to ports, and pins to pins. The complexity of Boolean graph matching and graph isomorphism is a nonissue for DTAS.

2.5 Knowledge-Based Design

The AI community studies the process of human design for the purpose of moving design into the machine (Mostow, 1985). During the 1980's, AI researchers demonstrated the utility of knowledge-based models of design for the synthesis task (Stefik, 1981; Brown and Chandrasekaran, 1986). Knowledge-based design systems have been applied in domains such as computer systems (McDermott, 1982), circuit boards (Birmingham and Siewiorek, 1988), integrated circuits (Steinberg, 1987), paper handling systems (Mittal, Dym, and Morjaria, 1986), and others.

One appeal of knowledge-based techniques for digital design is in the flexibility and naturalness of symbolic representations over analytic models. Another is in the use of domain knowledge to control heuristic search over tasks that are otherwise intractable by mathematical methods. The success of R1/XCON (McDermott, 1982) illustrated the potential for knowledge-based synthesis of digital systems. Brewer and Gajski (1986) and Wolf, Kowalski, and McFarland (1986) describe knowledge-based approaches to synthesis of very large scale integrated (VLSI) systems. The DAA system (Kowalski, 1985) uses domain knowledge to aid in the design of an IBM 370

at the micro-architecture level, and the SOCRATES system (de Geus and Gregory, 1986) uses rule-based search for technology mapping and optimization of Boolean logic.

Two knowledge-based design systems that have received considerable attention in the IC CAD community are VEXED (Mitchell, Steinberg, and Shulman, 1984) and MICON (Birmingham and Sieworek, 1984). VEXED is an interactive editor that assists designers traversing a hierarchy of function blocks to synthesize an NMOS implementation of a circuit. VEXED relies on a hierarchy of functions and constraint propagation (Kelly and Steinburg, 1987) to guide its refinement process. MICON synthesizes small computer systems from high-level specifications. MICON hierarchically refines an abstract system specification into a detailed configuration of microprocessors, memory chips, input/output devices, and bus drivers.

Like MICON, DTAS uses design knowledge that decomposes digital components from an abstract specification into configuration of physical parts. DTAS's decomposition methods are similar to MICON's design templates, each of which describe one level of decomposition. Unlike MICON, DTAS also supports search across alternative design styles; MICON is deterministic, i.e., the existence of design alternatives is defined as an error condition. Like VEXED, DTAS operates at the level of RT synthesis. VEXED, however, models the design process as

$$\text{Design} = \text{Top-Down Refinement} + \text{Constrain Propagation}$$

The problem with this model is that propagation of area and delay constraints is difficult without having reached the level of physical cells, at which point the area and delay characteristics of a circuit can be computed. While constraints can be used in selecting between design styles, this does not appear to be how they are used in VEXED.

In DTAS, constraint satisfaction is achieved by exploring alternative design styles and evaluating fully mapped designs. Another difference between VEXED and DTAS is in the level of detail found in the design knowledge. VEXED decomposes components using rules that leave the type and connections between components unspecified. While this is done for reasons of generality, it fails to recognize the fact that knowledge about circuit design is often very specific. One can make general characterizations about some classes of components, e.g., arithmetic components have a carry input and output, but such generalizations are difficult to realize in the design process; for instance, it is also necessary to mention the other ports of the component. DTAS uses design knowledge that is specific to classes of components in which all component connections are specified. DTAS allows generalization over the width of the ports and it provides a shorthand for defining methods for components with optional ports.

2.6 Knowledge Acquisition

Knowledge acquisition is the task of constructing a sufficiently complete and correct knowledge base to provide a high-level of performance. One of the major impediments to developing knowledge-based systems is the knowledge acquisition bottleneck. A commonly proposed solution is the construction and use of an automated knowledge acquisition tool. Examples include TEIRESIAS (Davis, 1981), ETS (Boose, 1984), MORE (Kahn, Nowlan, and McDermott, 1985), SALT (Marcus, McDermott, and Wang, 1985), SEAR (van de Brug, Bachant, and McDermott, 1986), KNACK (Klinker et al., 1987), and CGEN (Birmingham and Siewiorek, 1989). These tools typically interact with domain experts, organize the acquired knowledge, and generate expert rules.

For instance, CGEN is the knowledge acquisition component of MICON. CGEN acquires knowledge of how to build and when to use various computer structures–knowledge required by MICON–from interactions with hardware designers. CGEN provides designers with a graphics interface that allows them to describe computer structures with schematic drawings. CGEN queries designers for other pertinent information, uses design state information to constrain the schematic, generalizes the constrained schematic, and adds it to the knowledge base.

Machine Learning techniques have also been used for automating the acquisition of design knowledge. Techniques that improve system performance by shortening design and avoiding design flaws have received the greatest attention. This is referred to as *learning control knowledge.* Both inductive (Lathrop and Kirk, 1986) and analytic (Mitchell, Mahadevan, and Steinberg, 1985; Tong and Franklin, 1989) methods have been applied to this problem. There is also research that shows how acquiring too much control knowledge can hinder performance (Minton, 1988). Analytic methods have also been applied to the problem of learning to design novel artifacts (Mitchell, Mahadevan, and Steinberg, 1985); sometimes referred to as *learning implementation knowledge.* Huhns and Acosta (1988) applies learning to design by analogy to previous design experiences. These techniques typically learn design knowledge by analyzing the behavior of the design system or by generalizing the behavior of an expert designer.

For instance, LEAP is the learning apprentice system associated with VEXED. LEAP records the actions of an expert designer as he interacts with VEXED and then generates proofs that the functionality of the design satisfies the initial specification using Boolean algebra. The proofs are treated as explanations for the designer's actions and are generalized into rules that allow VEXED to take the same actions in similar situations.

LOLA is the knowledge acquisition subsystem for DTAS. LOLA acquires design knowledge as a preprocess to design synthesis. LOLA differs from MICON and other knowledge acquisition systems in that it does not query the user for this knowledge but uses analytic techniques to generate it. LOLA calls DTAS to synthesize test cases that it uses to evaluate and constraints the methods generated, as well as the generic methods already available to DTAS. LOLA differs from LEAP in that LEAP is intended to acquire technology-independent design knowledge while DTAS is intended to acquire knowledge that is specific to the use of cells in a given ASIC library.

Although much of the literature concerning knowledge acquisition stresses its role in knowledge maintenance, an issue that has yet to be clearly addressed is that of maintaining technology independence. The "fixed parts" assumption cited by Mittal and Frayman (1989) may explain why knowledge aquisition techniques have not been applied to this problem, i.e., if fixed parts can be assumed, then maintaining technology independence is a nonproblem. However, the fixed parts assumption is unrealistic. Case in point: R1/XCON (McDermott, 1982), a rule-based design system used by Digital Equipment Corporation to configure computer systems, has grown from 700 rules to over 6,200 rules with future growth expected. The primary reason indicated is that "*XCON must continually be updated to reflect new products and computing concepts...*" (Soloway, Bachant, and Jensen, 1987).

Thus, the distinction between LOLA and LEAP is an important one. Once captured, either by knowledge engineering or acquisition techniques, technology-independent design knowledge changes slowly, if at all. Technology-specific design knowledge, on the other hand, changes with the fabrication technology (which is quite frequent for the IC domain). Thus, the expected lifetime and utility of an acquisition system for technology-specific design knowledge should be greater than that of an acquisition system for technology-independent design knowledge.

Chapter 3
Approach

In this chapter, I describe the rationale and motivation behind my approach to component generation and technology adaptation. First, I present a view of synthesis as functional decomposition and search; then, I present a model of synthesis for component generation; finally, I describe the robustness problem as a failing of this model and outline an knowledge-acquisition approach to resolving this problem.

3.1 Functional Decomposition

Human designers overcome the problems that limit current approaches to component generation by viewing RT components as functional blocks, rather than as sets of Boolean equations. By applying a top-down design strategy, they use principles of logic design, such as those described by Mano (1979), to iteratively decompose high-level components into a hierarchy of increasingly smaller subcomponents. When there are alternative ways to decompose a component, knowledge of design styles is used to select the decomposition that best fits the design constraints. Decomposition stops when the design reaches a level of granularity that can be implemented with cells from a given layout library. When the functionality or I/O interface of a library cell does not precisely fit the design specification, design principles can again be applied to augment or modify the library cell until its function and form does match the requirements for implementation of the generic design.

For example, Figure 3.1 illustrates an instances of this design methodology applied to a 16-bit arithmetic component similar to that seen earlier in Figure 2.3. In Figure 3.1(a), the arithmetic unit is decomposed into a 16-bit adder with an external combinational circuit (CC_0) on its data and carry inputs. In Figure 3.1(b), the combination circuit is further decomposed into 16 identical circuits ($CC_{0.0}$), one for each data input, and another combination circuit ($CC_{0.1}$) controlling the carry input. Circuits $CC_{0.0}$ and $CC_{0.1}$ can be described by three simple Boolean equations and further synthesized from these using conventional logic synthesis techniques.

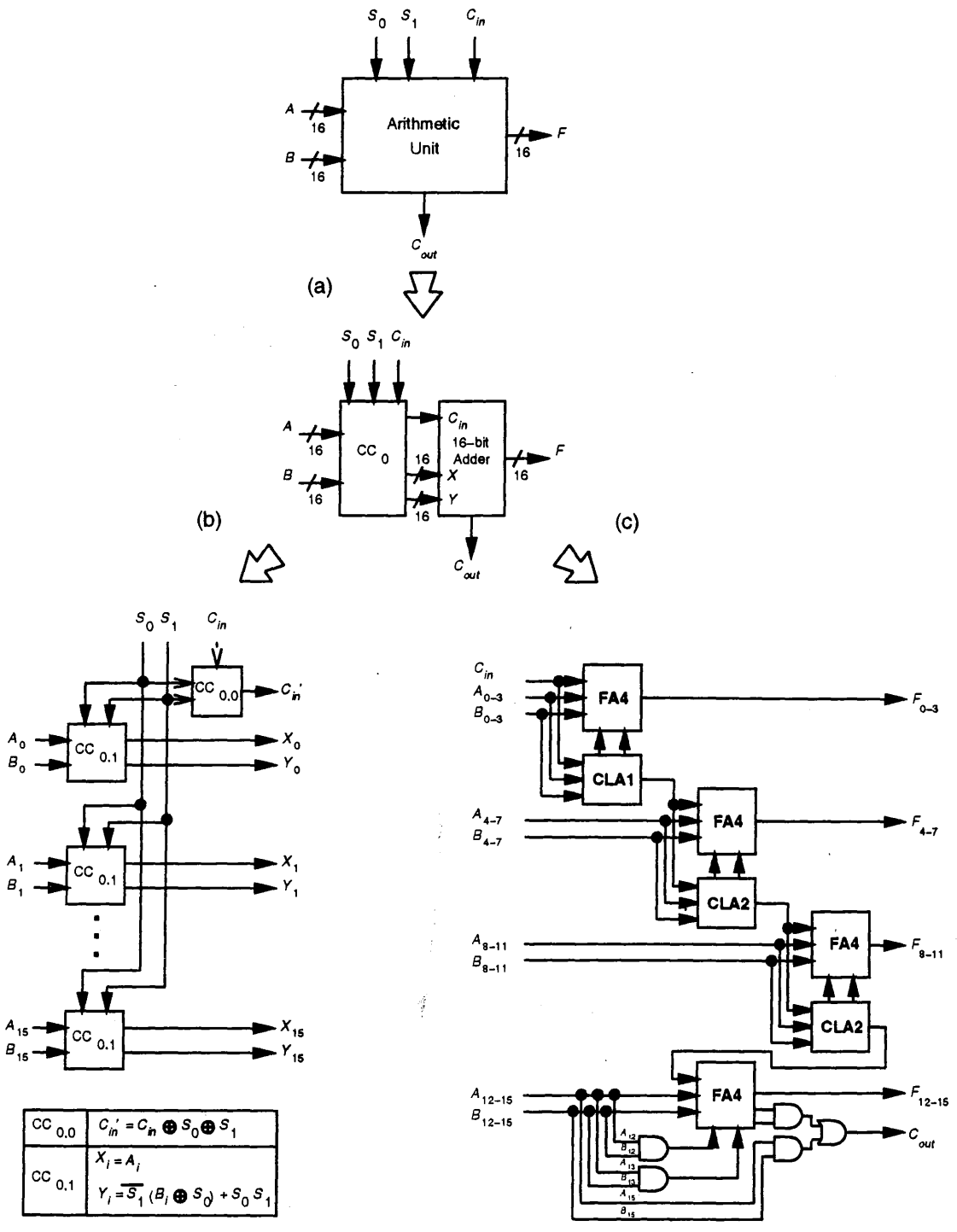


Figure 3.1: Functional decomposition of arithmetic unit: (a) decomposition of arithmetic unit; (b) decomposition of CC_0; and (c) decomposition of adder.

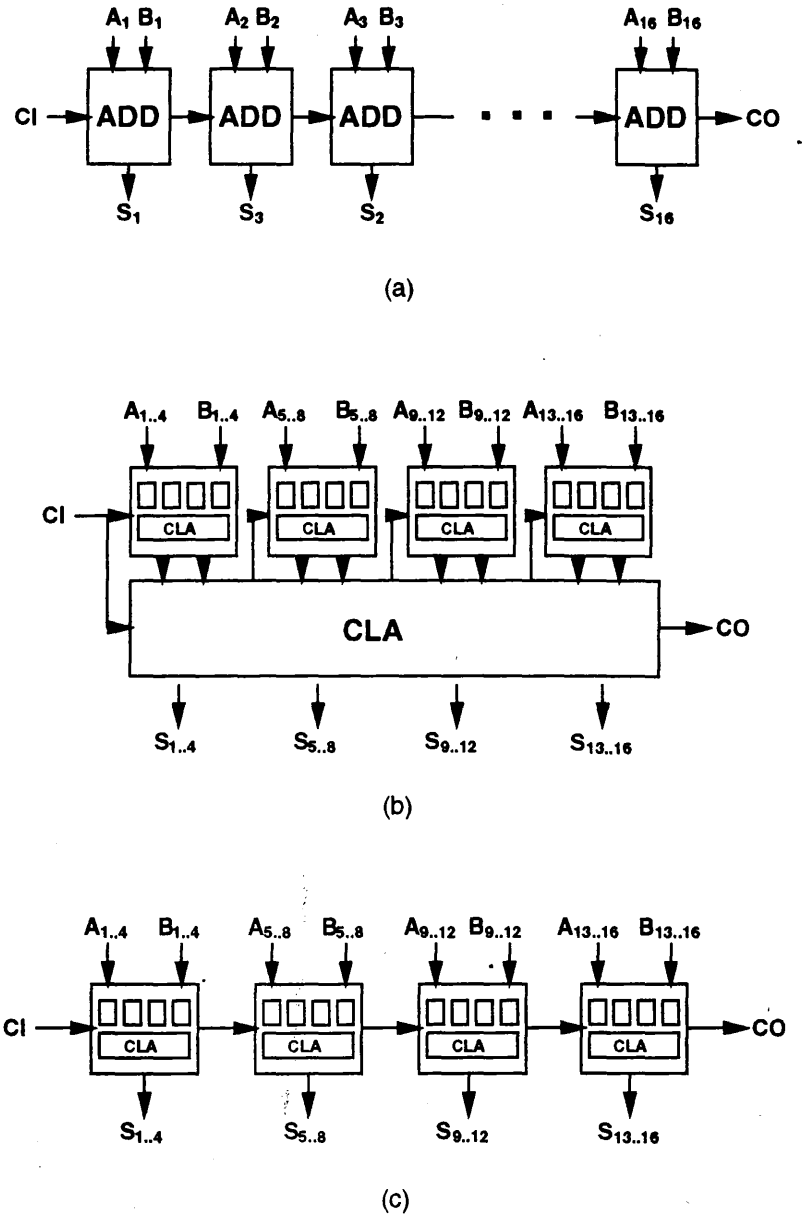


Figure 3.2: Alternative decompositions of 16-bit adder: (a) ripple-carry style; (b) carry look-ahead style; and (c) hybrid style.

The 16-bit adder can be decomposed using any one of a number of alternative designs styles, which are depicted in Figure 3.2. If area is a critical constraint, than the adder can be decomposed into 16 1-bit full adders using a ripple-carry style (Figure 3.2(a)). If delay is critical, then a carry look-ahead (CLA) style can be used, decomposing the adder into four 4-bit CLA adders plus a CLA generator (Figure 3.2(b)). It's also possible to mix styles, decomposing the adder into four 4-bit CLA adders whose carries are rippled (Figure 3.2(c)).

Assume for this example that the adder is being mapped into LSI Logic's macrocell library (LSI, 1987). This library happens to include a 4-bit adder cell (FA4) and two CLAs (CLA1 and CLA2). These cells have certain inputs and outputs that do not match the generic definition of a CLA adder exemplified in Figure 3.2(b). As shown in Figure 3.1(c), library-specific design knowledge is needed to connect these cells in order to implement the functionality of the desired 16-bit adder.

3.2 Synthesis as Search

In my approach to component generation, I view synthesis as search through a two-dimensional space of functionally-equivalent designs (Kipps and Gajski, 1990). Along one dimension, designs vary by their degree of functional abstraction; search moves from the most abstract design, i.e., the initial functional specification, to the most specific design, i.e., a netlist of library cells. Along the other dimension, designs vary in their structural configuration; search moves between designs at corresponding levels of abstraction, e.g., the level of library cells, looking for designs that conform to given performance constraints.

In traversing the design space, Gajski and Brewer (1986) identify three issues that must be addressed by a synthesis tool: *style selection*, *technology mapping*, and *optimization*. Style selection is the problem of choosing appropriate design styles for meeting design constraints (e.g., serial vs. parallel implementation). Technology mapping is the problem of selecting a physical instantiation that provides the required functionality and meets performance constraints. Optimization is the problem of applying functionally equivalent structural transformations to the design in order to meet violated constraints. Style selection and technology mapping correspond to search along the dimension of abstraction; optimization corresponds to search along the dimension of configuration.

Approaches to component generation that rely upon logic synthesis are primarily concerned with optimization, focusing search on the dimension of configuration. Little search effort is spent on style selection and technology mapping. In my approach, I focus search primarily on style selection and technology mapping. I rely upon existing

logic synthesis technology for optimization of the portions of a design that must be decomposed to the level of Boolean logic.

3.3 Functional Specification

Any approach to component generation requires a data model for representing the functionality of components and cells. As described in Chapter 2, the model used by existing approaches is that of Boolean logic. In my approach, I model components and cells using a symbol-system representation that I refer to *functional specification.*

With functional specification, a component or cell is described by the abstract digital function it performs and by the name and widths of its input and output ports; additional features, such as operations computed or area and delay characteristics, are described with attribute/value pairs. For instance, a 4-bit adder is described as computing the function ADD and as having two 4-bit data inputs, a 1-bit carry input, a 4-bit data output, and a 1-bit carry output. If the adder is a component to be designed, this description might be augmented with an attribute/value pair that indicates the design style, such as RIPPLE or CLA. On the other hand, if the adder is a library cell, this description might be augmented with an attribute/value pair that indicates its area or delay or the load on its inputs.

One appeal of this method of description is its similarity to how a human designer describes components. For instance, if a digital design engineer were given the description of the adder outlined above, he would understand it and have no problem generating a design. In other words, human designers have an abstract language for describing and decomposing functional components and this language is not Boolean logic.

3.4 The Derivational-Process Model

I have developed the derivational-process model as an approach to synthesis that reflects the top-down design process outlined above. This model extends the traditional view of component generation and logic synthesis in several unique ways, the four most significant of which include: the addition of a top-down design phase, *functional decomposition*; the use of *design styles* and *derivation-driven search* to generate a candidate set of designs that closely satisfy design constraints; a *model of technology adaptation* to ensure robustness to library changes; and the instantiation of *principles of logic design* to aid in acquiring both design and mapping knowledge.

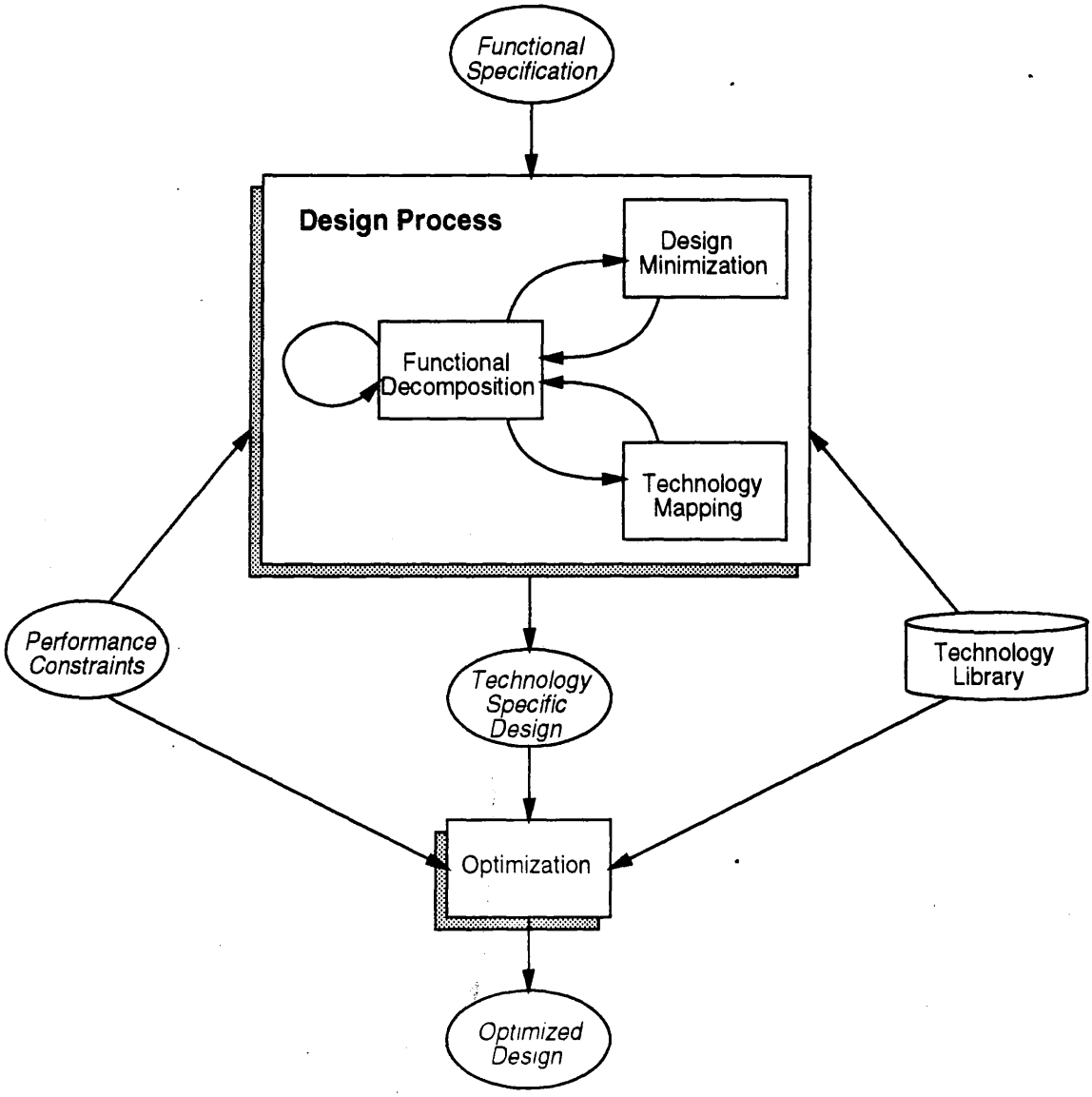


Figure 3.3: Derivational-process model.

As depicted in Figure 3.3, synthesis in the derivational-process model is factored into a *design phase* and an *optimization phase.* The design phase is further factored into interacting processes of *functional decomposition*, *design minimization*, and *technology mapping.* The purpose of these processes is summarized below.

- **Functional decomposition** is a process of top-down design. Selecting a design style appropriate to constraints, the functional decomposition process outputs a hierarchical netlist of connected subcomponents. Some subcomponents can be described by Boolean equations; these are passed to the design minimization process. Others are functionally "close" to library cells, in which case they are treated as "leaf" nodes and passed to the technology mapping process.
- **Technology mapping** is a process that instantiates the leaf nodes of the generic design with configurations of library cells. In simple cases, this merely involves a one-to-one mapping. In other cases, the technology mapping process implements the leaf node with library cells augmented by generic subcomponents. The generic subcomponents can be passed to the functional decomposition process for further refinement and design.
- **Design minimization** is a process that subsumes a variety of graph reduction processes, such Boolean minimization, state reduction, and protocol analysis. Eventually the design phase outputs a physical design, implemented with ASIC library cells, which is then passed to the physical optimization phase for fine-tuning.
- **Physical optimization** is a process of refining the physical design along critical paths in order to meet performance constraints. Note that this phase of the derivational process model is oversimplified. The current focus in developing the derivational-process model is on extended capabilities afforded by functional decomposition and technology mapping. The optimization phase can be extended to deal with RT cells as future research. The implementation of this model relies upon external logic synthesis tools to support optimization.

The input to the model is a set of functional specifications for generic hardware components. Additional inputs include performance constraints on the design and a library of physical cells. The output is an optimized netlist of library cells that constitutes a physically-realizable design.

The processes of functional decomposition and technology mapping are controlled by *derivation-driven search*, the goal of which is to generate a set of candidate library-specific designs whose physical characteristics either meet or approximate the given design constraints. Derivation-driven search uses design knowledge to explore the space of generic and physical designs along the dimension of abstraction. Individual elements of design knowledge can be represented with tuples that pair component specification patterns with bodies of actions that implement the specified components by generating netlists of connected subcomponents.

There are two types of design knowledge, distinguished by their role in the design process, i.e., whether they are used during functional decomposition or technology mapping. Elements of design knowledge for functional decomposition are called *decomposition methods.* Decomposition methods define how to achieve the functionality of a generic component in terms of connected subcomponents. Methods encapsulate design styles for implementing a class of components. Several methods can be applicable to the same component, reflecting alternative design styles.

Elements of design knowledge for technology mapping are called *construction methods.* Construction methods define how to implement a generic component by augmenting the functionality of a physical library cell. Decomposition methods can be viewed as operating from the top-down, designing increasingly specific components, while construction methods can be viewed as operating from the bottom-up, designing increasingly abstract components. Eventually, the most specific decomposition methods merge with the most abstract construction methods, denoting the transition from functional decomposition to technology mapping.

3.5 Synthesis Example

To illustrate the use of decomposition and construction methods in derivation-driven search, consider the example below in which we synthesize the design of an arithmetic logic unit (ALU). To keep the explanation simple, the methods are depicted graphically. Likewise, details of the search-control strategy are also ignored. The purpose of this example is merely to demonstrate how the use of search, functional decomposition, and technology mapping can extend synthesis to regular-structured components and improve design quality.

Assume the example ALU can perform a set of basic arithmetic, comparison, and logic operations. A and B are n-bit data inputs, combined to generate an operation at output F. The function-select lines S distinguish the operation. Carry input C_{in} and carry output C_{out} are only useful during arithmetic operations. Output R carries the results of comparison operations.

The methods in Figure 3.4 represent two alternative design styles: an integrated design style is depicted in Figure 3.4(a), and a segregated style is depicted in Figure 3.4(b). The central component in the integrated style is an adder (ADD) with carry enable C_E. By controlling the inputs and outputs of the adder with various external combinational logic units (CC), it can be made to perform the required arithmetic and comparison operations; by disabling the carry, it can be made to perform all sixteen logic operations on two operands. The segregated style separates the arithmetic and comparison operations, which again require an adder, from the logic

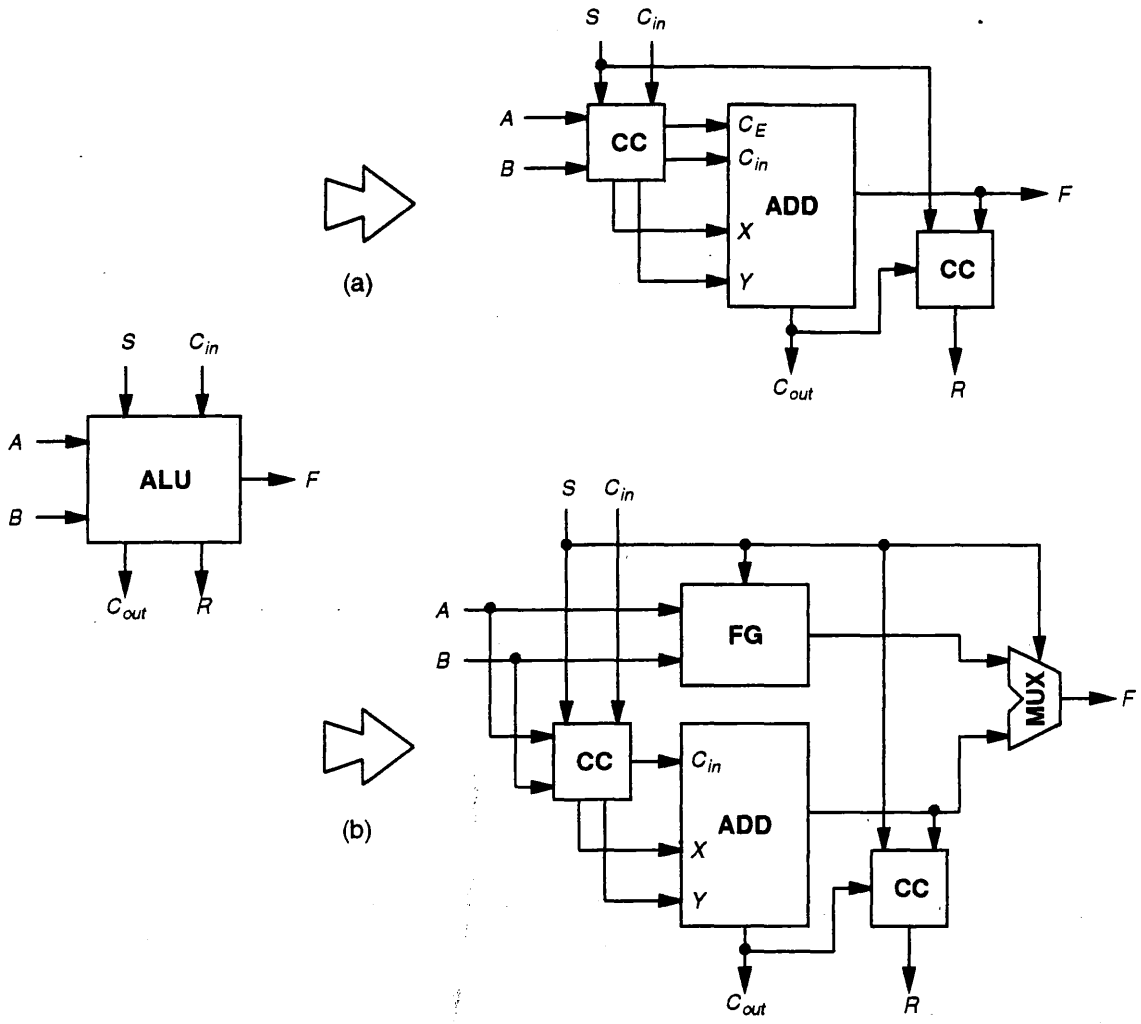


Figure 3.4: Alternative decomposition methods for ALU: (a) integrated style; and (b) segregated style.

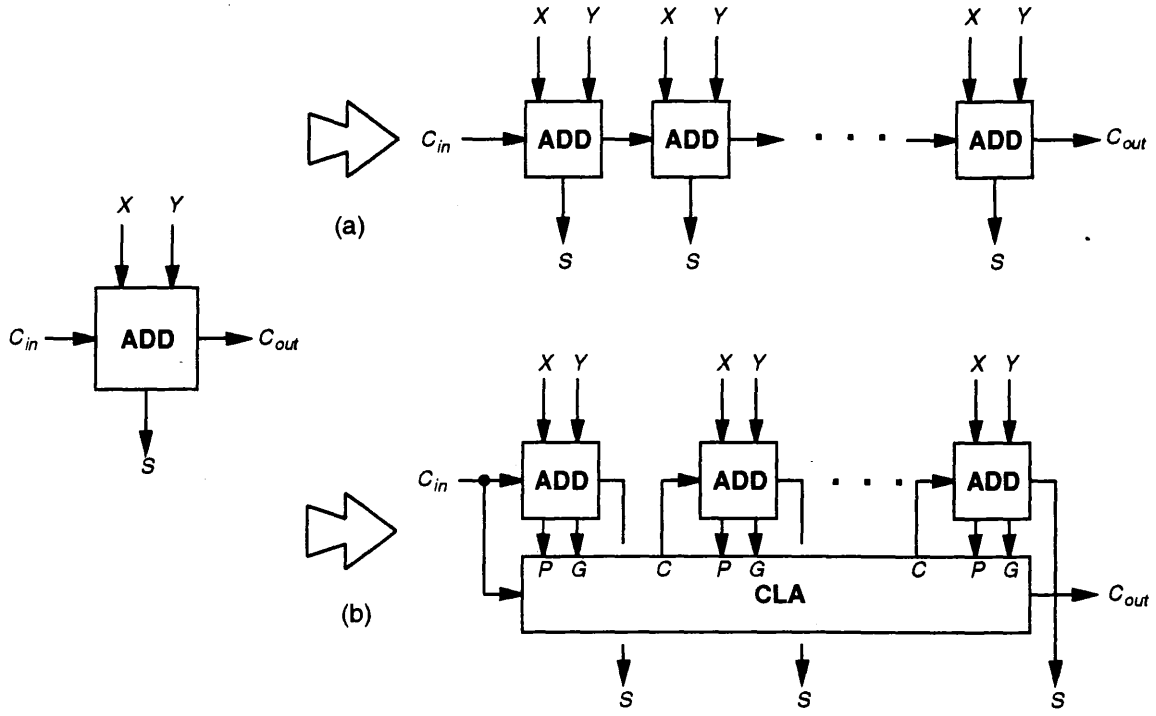


Figure 3.5: Alternative decomposition methods for adder: (a) ripple-carry style; and (b) carry look-ahead style.

operations, which only require a function generator (FG). The appropriate operations are selected by passing the outputs of these two components through a multiplexer (MUX).

The methods in Figure 3.5 also depict two alternative design styles. The first method, Figure 3.5(a), depicts the ripple-carry style in which the carry output of each component adder is attached to the carry input of the next. The second method, Figure 3.5(b), depicts a carry look-ahead style in which a carry look-ahead generator (CLA) is used to compute the carry inputs to component adders. (The details of generating the combinational logic needed for the CC's as well as the logic used for the function generator FG, multiplexer MUX, and carry look-ahead generator CLA are not important to this example.)

The advantage of using an integrated style in designing an ALU is that it integrates the combinational logic needed for the arithmetic and logical operations, eliminating redundant logic as well as a level of delay required for multiplexing. The advantage of a carry look-ahead style in designing an adder is that it reduces propagation delay, although it increases the size of the circuit. An adder can be decomposed

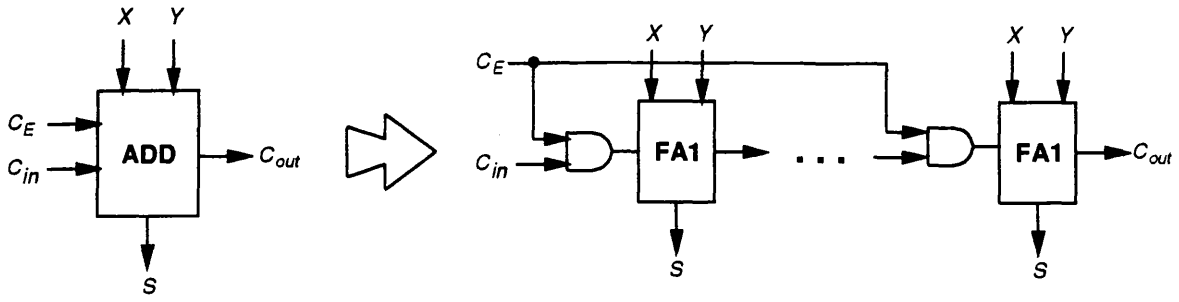


Figure 3.6: Construction method for adder with carry enable C_E.

into several levels of adders, so its design can actually use a hybrid ripple-carry/carry look-ahead style.

Construction methods come into play by providing the synthesis tool with knowledge of available library components and methods for their use. Suppose the target layout library includes four different adders: a 16-bit adder (FA16), a 4-bit adder (FA4), a 2-bit adder (FA2), and a 1-bit adder (FA1). Given the ALU decomposition methods seen above, there are two types of n-bit adders needed in designing an ALU: one with a carry enable (C_E), and one without. Because none of the library adders come with a carry enable, the former can only be implemented from n 1-bit adders, augmented by an AND gate on its carry input, as depicted in Figure 3.6. The latter can be constructed from any of the existing library adders, depending on the size of n, as shown by the methods in Figure 3.7.

Now assume that we wish to synthesize a 32-bit ALU. Having no knowledge of the complex library components, a component-generation tool would probably use an integrated style to design the ALU, decomposing the design to the level of Boolean gates. Unless time or area were extremely critical, a hybrid style might be selected in designing the adder. Given library-specific design knowledge in the form of the construction methods depicted in Figures 3.6 and 3.7, it is possible to generate designs into functional library cells and, from the performance values associated with each cell, to accurately measure the trade-off between design alternatives. With the integrated style, the most complex library component that can be used is the FA1, while with the segregated style it is possible to use any of the library adders. Given that two FA16's sufficiently outperform 32 FA1's, then the component generation tool would prefer the segregated design style for ALUs to the integrated and output a higher quality design than otherwise possible.

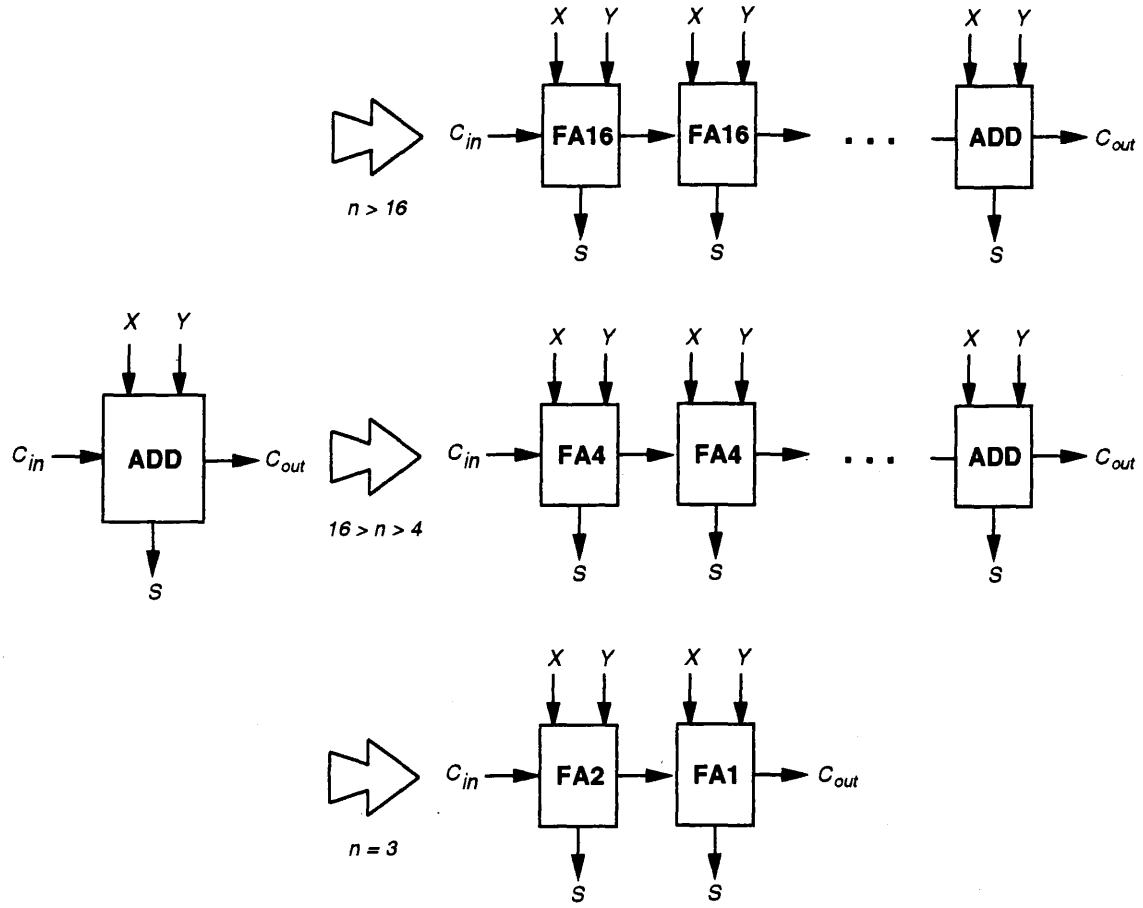


Figure 3.7: Construction methods for adder without carry enable.

3.6 The Robustness Problem

The use of functional layout cells and technology-specific design knowledge can improve the state of the art in component generation, but only at a cost to robustness against technology changes. The effectiveness of the derivational-process model of design synthesis can be measured against existing approach to component generation in terms of competence, quality, and robustness.

Because the derivational-process model scales up to regular-structured microarchitecture components, it will have improved competence. Because it considers alternative design styles and complex library cells, it will have improved design quality. However, because this model utilizes library-specific design knowledge, it falls short in

regards to robustness to library changes or to changes in the fabrication technology. I refer to this shortcoming as the *robustness problem.*

The robustness problem is a special case of the knowledge acquisition bottleneck encountered in developing expert systems. The AI literature reports many efforts to reduce cost and increase performance of knowledge-based systems with semi-automated tools for aiding the knowledge acquisition process. For instance TEIRESIAS (Davis, 1982) interactively repairs and extends the knowledge base of the medical diagnostic system MYCIN (Shortliffe, 1976); LEX (Mitchell, Utgoff, and Banerji, 1983) learns search control heuristics for problem solving in integral calculus; LEAP (Mitchell, Mahadevan, and Steinberg, 1985) learns new design methods for the VEXED (Steinberg, 1987) IC design system; and CGEN (Birmingham and Siewiorek, 1989) interactively captures design expertise for the MICON (Birmingham, Gupta, and Siewiorek, 1989) computer design system.

As with the above systems, my approach to alleviating the robustness problem has been to develop an adjunct model of knowledge aquisition for technology adaptation that I call *technology compilation.* The purpose of this model is to acquire new construction methods given knowledge of the principles of logic design, specifications of available cells in the ASIC layout library, and knowledge of the component classes for which the synthesis system is accountable. This model is intended as a preprocess to synthesis, generating methods without actually having run the component generator, essentially bootstrapping the component generator into the new cell library.

3.7 Principles of Logic Design

When presented with new library cells, human designers adapt quickly. They do so by bringing to bear bits of knowledge and techniques for logic design that are essentially technology and application independent. This knowledge is what I refer to as the *principles of logic design.*

The principles of logic design govern how the designs of digital components can be organized, decomposed, implemented. These principles include:

- **Factoring**. Like components are combined in a tree to implement components with a wider input width. This principle is appropriate for logic components, such as Boolean gates and multiplexers.

- **Sequencing**. Like components are cascaded or replicated in sequence, again to implement components with a wider data width. This principle is appropriate for arithmetic components, such as adders, comparators, and counters, as well as logic components, such as function generators.
- **Multiplexing**. Dissimilar components are combined through a multiplexer or buss. This principle is appropriate for implementing multiple operation components, such as arithmetic logic units (ALUs).
- **Exclusion**. Component inputs and outputs can be suppressed. This principle is appropriate for implementing components with a narrower data width or with less functionality. For instance, implementing 3-bit multiplexer with a 4-bit multiplexer.
- **Externalization**. Component inputs and outputs can be augmented by external combinational logic. This principle is appropriate for implementing multiple operation components. For instance, implementing an adder/subtractor with an adder and external control logic.

Applications of exclusion, sequencing, externalization, and multiplexing are demonstrated by example in Figure 3.8. For this example, assume that the physical cell is a simple 4-bit ALU (ALU4) with four operations: addition (ADD), subtraction (SUB), AND, and OR.

To generalize the data width of the ALU4 so as to implement a generic n-bit ALU with the same four operations, the principles of exclusion and sequencing can be applied. When $n < 4$, as in Figure 3.8(a), the ALU4 can be used to implement the n-bit ALU by setting the $4 - n$ least significant input pins to low and grounding the $4 - n$ least significant output pins. When $n > 4$, as in Figure 3.8(b), the ALU4 can be used to implement the n-bit ALU by cascading $\lfloor \frac{n}{4} \rfloor$ ALU4's, rippling carries and attaching a generic mod$(n, 4)$-bit ALU at the tail. To generalize the functionality of the ALU4, the principles of externalization and multiplexing can be applied, augmenting I/O when the change in functionality is slight and separating functionality when a more complicated change is required. In Figure 3.8(c), the ALU4 is generalized to implement an ALU with relational operators (R) by augmenting the select line, data output, and carry output with an external combinational circuit (CC). In Figure 3.8(d), the ALU4 is generalized to implement an ALU with all 16 logic operations. Since this marks a substantial change from the functionality of the ALU4 it is not done by augmentation but by coupling the ALU4 to a function generator (FG) and passing the outputs through a multiplexer (MUX).

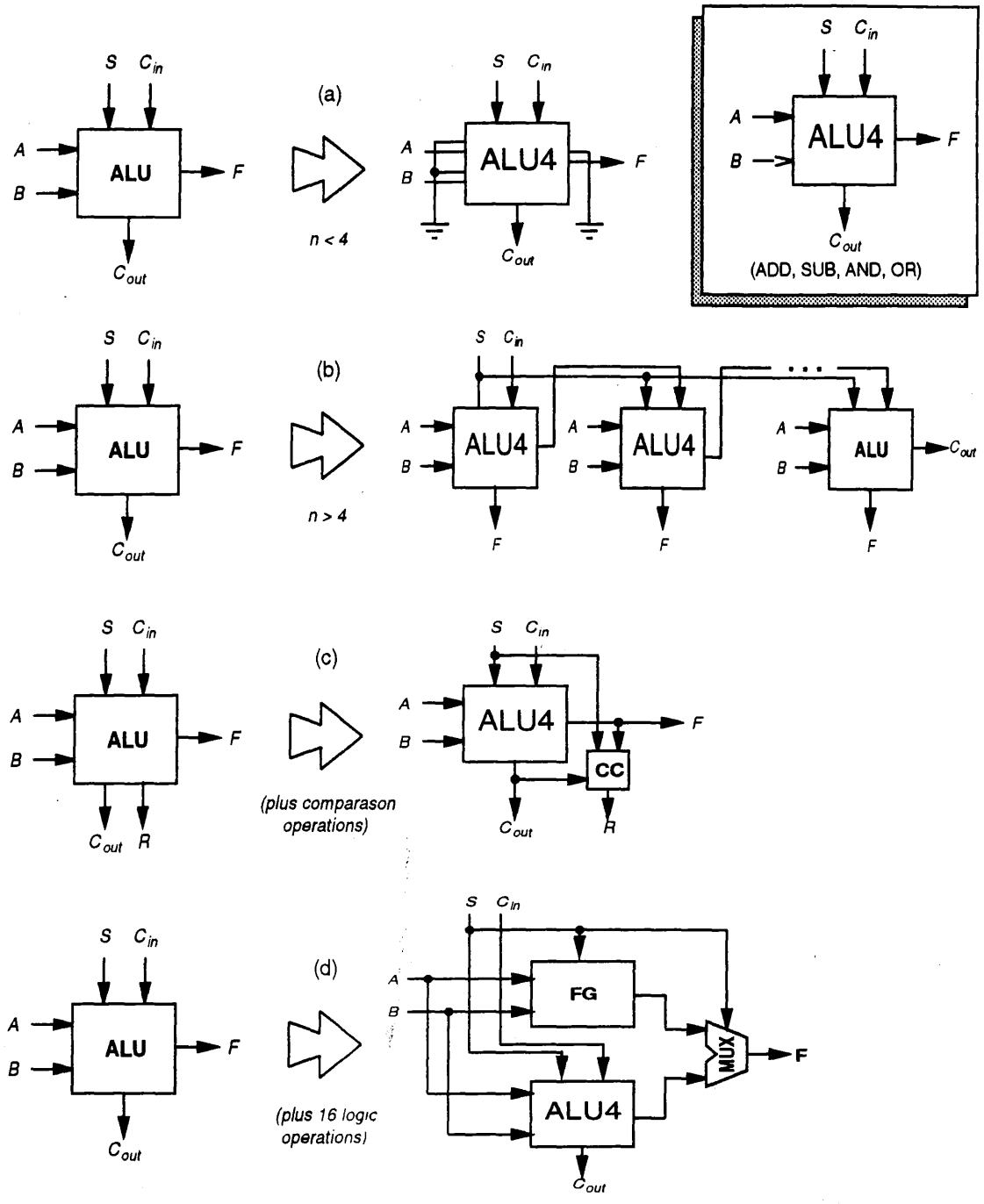


Figure 3.8: Principles of logic design: (a) exclusion; (b) sequencing; (c) externalization; and (d) multiplexing.

3.8 Technology Compilation

The objective of technology compilation is to compile a set of technology-specific (i.e., library-specific) design knowledge in the form of construction methods. The relationship of technology compilation to design synthesis is depicted in Figure 3.9. Technology compilation is intended to run in advance of synthesis.

As depicted in Figure 3.9, technology compilation operates in a cyclic manner, drawing on the principles of logic design to generate construction methods specific to a given layout cell library. In the *instantiation* phase, the available library cells are compared to the class of generic components represented in the decomposition methods. If a library cell is "close" in functionality to a generic component and there is a design principle for implementing the generic component in terms of the library cell, then the principle is compiled into a construction method for implementing the generic component in terms of the library cell. The type and attributes of the generic component as well as literal attributes of the cell, such as the widths of its data ports, are used to instantiate the design principle, creating a construction method particular to the component and cell. This method is added to the construction methods being generated for the cell library.

In the *evaluation* phase, experiments are run to compare the newly acquired construction method with the existing methods, both library-specific and generic. The phase is looking for conditions under which certain methods do not generate designs that are of inferior quality with regard to all measured performance characteristics, such as area and delay, to the designs generated by other methods.

In the *constraint* phase, the conditions detected during evaluation are turned into constraints that can be attached to the errant method and tested during synthesis. These constraints keep methods from being applied under conditions when they are known to generate designs that are of lesser quality than alternative methods.

As an example, consider the situation where the cell library contains 1-, 2-, 4-bit adder cells and a decomposition method for implementing n-bit adders using a carry look-ahead style to level of Boolean gates. Assume also that it is only possible to generate a construction method that ripples the library adders. In this situation, it might be the case that for n-bit adders, where n is less than 32, rippling the 4-bit library adders might give the smallest and fastest designs. However, there will be a limit on the size of n after which a Boolean carry look-ahead adder will provide a reduced delay over a ripple-carry adder, even one using 4-bit library adders. This value of n can be used as a constraint on the decomposition method.

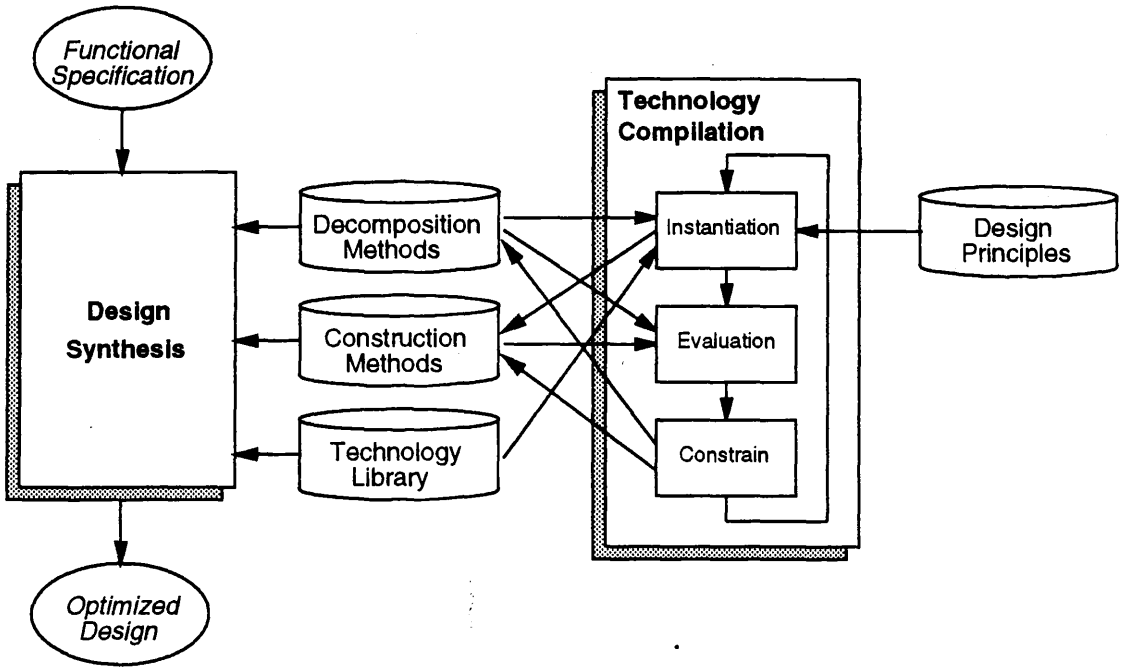


Figure 3.9: Technology compilation and design synthesis.

Chapter 4
The Component Decomposition Algorithm

In this chapter, I present a formal definition of the component decomposition algorithm. First, I give a conceptual overview of the algorithm; then, I define the algorithm and its data structures; finally, I explain the search control strategy applied by the algorithm. Examples are provided to clarify concepts as needed; an annotated example is provided in Chapter 5.

4.1 Overview

The component decomposition algorithm performs a depth-first expansion of the design space for an input netlist of uninstantiated components or *modules*. It returns a list of alternative implementations of the input netlist, each of which is fully mapped to cells from a target ASIC library. The basic operation of the algorithm is to find all alternative implementations of the modules in the input netlist and to return a netlist for each combination of these alternatives.

The alternative implementations of a module are generated recursively. The algorithm first compares the functional specification of the module to the cells in the given library. Each matching cell is one alternative implementation. The algorithm then compares the specification to the defined decomposition methods. Each applicable method is expanded, resulting in a netlist of uninstantiated subcomponents. The algorithm recursively finds all alternative implementations of each netlist, all of which are also alternative implementations of the module. This process continues until all subcomponents have been decomposed to the point that further decomposition is not possible. Netlists containing subcomponents for which no fully-mapped implementation exists are pruned from the design space.

The component decomposition algorithm is intended to find "desirable" designs by mixing design styles (as encapsulated in decomposition methods) and by mapping to configurations of complex library cells (using construction methods). As the number of methods and library cells increases, so does the the number of alternative designs and the quality of desirable designs. Unfortunately, the number of alternative designs grows exponentially, quickly making complete enumeration of the design space computationally intractable. The algorithm is defined such that the size of the design space is constrained by two search control principles.

The first principle constrains the ways in which alternative implementations of a netlist's modules can be combined. This principle says that identically specified modules in the same netlist must have like implementations. For instance, consider an input netlist of four 1-bit adders; assume there are two alternative adder implementations: a 1-bit adder cell and a netlist implementation of Boolean gates. There are 2^4 possible combinations of adder implementations to adder modules in the input netlist. According to the first principle, only two of these combinations are legal, namely, the combination in which all four adder modules are implemented by ASIC adder cells, and the combination in which all four adder modules are implemented by a netlist of Boolean gates.

The second search control principle constrains the number of fully-mapped alternative implementations that can be passed back up the design hierarchy. This principle applies a user-defined performance filtering function to the fully-mapped implementations of a module. Since the implementations are fully mapped, accurate performance estimates can be computed for each. The filtering function is expected to compare each of alternative implementation and to return a list of implementations it finds "acceptable." For instance, if area were critical, the filtering function could return some percentage of the smallest implementations; if speed were critical, it could return some percentage of the fastest. Taken together, these two search principles enable the component decomposition algorithm to expand otherwise computationally unenumerable design spaces and to generate a range of desirable implementations.

4.2 Data Structures

There are five principal data structures used in the component decomposition algorithm: *netlists*, *modules*, *cspecs* (component specifications), *cimpls* (component implementations), and *methods* (decomposition and construction methods). These structures and their fields are outlined in Figure 4.1. Angle brackets (< ··· >) delimit tuples, square brackets ([···]) delimit lists, and curly braces and bars ({ ··· | ··· }) delimit disjunctive terms.

Structure	Representation
netlist	<*ipins, opins, wires, cells, cspecs, modules*>
ipins	[*pin**]
opins	[*pin**]
wires	[*wire**]
cells	[<*name, cspec*>*]
cspecs	[<*name, cspec*>*]
modules	[<*name, module*>*]
module	<*ipins, opins, cspec, cimpl*>
ipins	[*pin**]
opins	[*pin**]
cspec	*cspec*
cimpl	*cimpl*
cspec	<*type, iports, oports, attrs*>
type	*symbol*
iports	[*port**]
oports	[*port**]
attrs	[<*name, value*>*]
cimpl	{ <*cspec, name, props*> | <*cspec, netlist*> }
cspec	*cspec*
name	*symbol*
props	[<*name, value*>*]
netlist	*netlist*
method	<*head, test, body*>
head	*cspec (pattern)*
test	[*cond*]
body	[*action**]
pin	*symbol*
wire	<*name, src, snk*>
src	*pin*
snk	[*pin*+]
port	<*name, width*>
name	*symbol*
width	*integer*
value	{ *number* | *symbol* | *list* }
list	[*value**]

Figure 4.1: Principal data structures.

4.2.1 Netlists and Modules

A netlist represents a configuration of connected component instances (*modules*). A netlist and its modules have a set of input pins (*ipins*) and output pins (*opins*) that represent the internal interface points of the netlist and the external interface points of each module. The pins of a netlist and its modules are connected by *wires*. Each wire has a single source (*src*) and one or more sinks (*snk*). The source of a wire can be an input pin of a netlist or an output pin of a module; conversely, a sink can be an output pin of a netlist or an input pin of a module.

The functionality of a module is defined by a component specification (*cspec*), while the design of a module is defined by a component implementation (*cimpl*). The specification of a module and its implementation are directly related. Each component specification is paired with an arbitrary number of component implementations. A component implementation represents either a single cell from an ASIC library or a netlist of modules. A module will be an instance of one of the implementations paired with its component specification. Before an implementation is assigned to a module, the module is said to be *uninstantiated.* A module is said to be *fully mapped* if it is an instance of a fully-mapped component implementation; a component implementation is fully mapped if it represents a physical library cell or if it represents a fully-mapped netlist; and, recursively, a netlist is fully mapped if each of its modules is fully mapped.

The relationship between netlists, modules, component specifications, and component implementations is depicted in Figure 4.2. This graph represents the hierarchical design space of the component decomposition algorithm. The component decomposition algorithm expands this space systematically in a depth-first manner. Distinct paths through the design space represent alternative hierarchical designs.

As shown in Figure 4.1, a netlist is represented with a tuple

<*ipins*, *opins*, *wires*, *cells*, *cspecs*, *modules*>.

The *ipins* and *opins* fields contain ordered lists of pins, denoting the input and output pins of the netlist. Each *pin* is represented with a symbol. The symbol identifies the pin; it must be unique among the pins of the netlist and its modules. The *wires* field contains a list of wires, denoting the connections between pins of the netlist and its modules. Each *wire* is represented with a triple <*name*, *src*, *snk*>, where *name* is a symbol identifying the wire, *src* is a pin (an input pin of the netlist or an output pin of one of the netlist's modules), and *snk* is a list of pins (output pins of the netlist and input pins of its modules).

The *cells* and *cspecs* fields of a netlist contains a list of pairs <*name*, *cspec*>, denoting the named component specifications that define the functionality of the netlist's modules. Each named component specification listed in the *cells* field corresponds to a library cell that has the same name and specification. A module whose

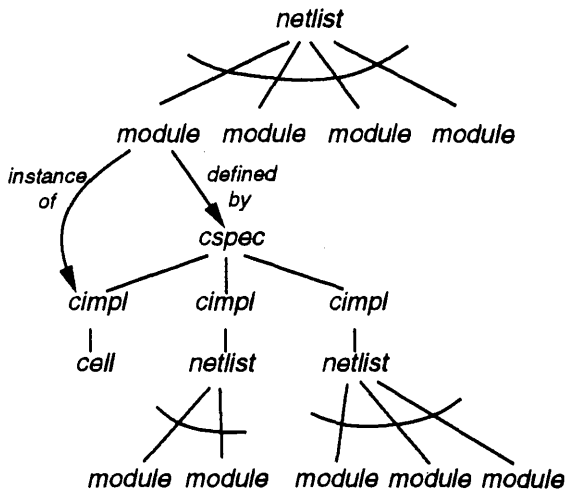


Figure 4.2: Design space representation.

functionality is defined by an entry in the *cells* field will be an instance of the corresponding library cell. On the other hand, a module whose functionality is defined by an entry in the *cspecs* field can be an instance of any of the specification's corresponding implementations. The same component specification can appear in both the *cells* field and the *cspecs* field. Finally, the *modules* field contains a list of pairs <*name*, *module*>, denoting the named modules of the netlist.

A module is represented with a tuple

<*ipins*, *opins*, *cspec*, *cimpl*>.

The *ipins* and *opins* fields are ordered lists of pins. The *cspec* field identifies the component specification that defines the functionality of the module. The *cimpl* field, if not empty, is an implementation of the component specification. The module is an instance of this implementation.

4.2.2 Component Specifications and Implementations

A component specification (*cspec*) is represented with a tuple

<*type*, *iports*, *oports*, *attrs*>.

The *type* field is a symbol that identifies the function of the component. Typical values for *type* include AND, OR, XOR, MUX, ADD, COMPARE, ALU, and MULT, which denote the purpose of the component as a functional unit. The *iports* and *oports* fields are ordered lists of ports, denoting the input and output ports of the

component. Each port is represented with a pair <*name*, *width*>, where *name* is a symbol identifying the port and *width* is a positive integer defining the port's width. The *attrs* field contains a list of pairs <*name*, *value*>, denoting the attributes of the component. A specification's attributes define particular aspects of the component, such as the operations to be computed or the design style to be used. The *name* of an attribute is a symbol; the *value* can be a number, symbol, or, recursively, a list of values.

A component implementation (*cimpl*) is represented with one of two forms: a triple <*cspec*, *name*, *props*>, which denotes a library cell implementation; and a pair <*cspec*, *netlist*>, which denotes a netlist implementation. In both cases, *cspec* identifies the component specification being implemented. In the first form, *name* is the symbolic name of a cell from the target ASIC cell library that implements the specified component; *props* contains a list of pairs <*name*, *value*>, denoting the physical properties of library cells, such as its area, delay, and power costs. These properties can be used to compute accurate cost and performance estimates for a fully-mapped netlist, which can subsequently be used in evaluating and comparing alternative designs. In the second form, *netlist* describes one level of decomposition of the component specified by *cspec*.

4.2.3 Representation Example

Several examples of the data structures described above are shown in Figure 4.3, which together depict the specification and implementation of a 1-bit full adder. Figure 4.3(a) shows three component specifications. The first specification, labeled `ADD.0` defines a half adder; it has type `ADD`, two 1-bit input ports, `I0` and `I1`, two 1-bit output ports, `O0` and `COUT`, and no attributes, as denoted by the empty list (`[ ]`). The second specification, labeled `ADD.1`, defines a full adder, as indicated by the extra 1-bit input port `CIN`. This specification has one attribute `<STYLE,HALF-ADDER>`, which, for this example, indicates the design style to be used in implementing the adder. The third specification, labeled `OR.0`, defines a 2-input OR gate.

Figure 4.3(b) shows a module that is defined by the component specification `ADD.1` from Figure 4.3(a). This module has three input pins (`[p0_00,p0_01,p0_02]`) and two output pins (`[p0_03,p0_04]`). Its *cimpl* field is empty, so the module is uninstantiated. Figure 4.3(c) shows three component implementations. The first two, labeled `HA1` and `OR2`, represent physical library cells, a half adder and a 2-input OR gate; the third represents a netlist implementation of a full adder.

```
ADD.0:  <ADD, [<I0,1>,<I1,1>], [<O0,1>,<COUT,1>], [ ]>
ADD.1:  <ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[<STYLE,HALF-ADDER]>
OR.0:   <OR, [<I0,1>,<I1,1>], [<O0,1>], [ ]>
```

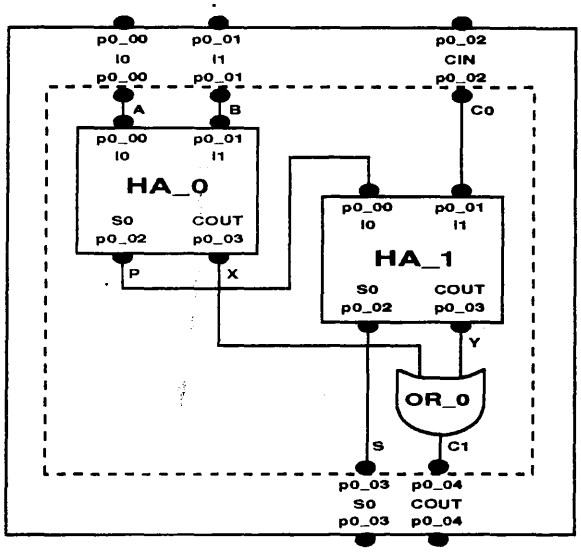
(a)

```
<[p1_00,p1_01,p1_02], [p1_03,p1_04], ADD.1, >
```

(b)

```
HA1:    <ADD.0, HA1, [<LOAD,...>,<AREA,...>,<DELAY,...>]>
OR2:    <OR.0, OR2, [<LOAD,...>,<AREA,...>,<DELAY,...>]>
ADD.1.0: <ADD.1, <[p0_00,p0_01,p0_02],
                  [p0_03,p0_04],
                  [ <A,p0_00,[p1_00]>,<B,p0_01,[p1_01]>,<C0,p0_02,[p2_01]>,<S,p2_02,[p0_03]>,
                    <C1,p3_02,[p0_04]>,<P,p1_02,[p2_00]>,<X,p1_03,[p3_00]>,<Y,p2_03,[p3_01]> ],
                  [ <HA1,ADD.0>],
                  [ <OR,OR.0> ],
                  [ <HA_0, <[p1_00,p1_01], [p1_02,p1_03], ADD.0, HA1>>,
                    <HA_1, <[p2_00,p2_01], [p2_02,p2_03], ADD.0, HA1>>,
                    <OR_0, <[p3_00,p3_01], [p3_02], OR.0, OR2>>]>  >
```

(c)



(d)

Figure 4.3: Example structures for 1-bit full adder: (a) component specifications; (b) module; (c) component implementations; and (d) graphical netlist for 1-bit full adder.

The netlist implementation in Figure 4.3(c) uses two half adder modules (HA_0 and HA_1) and an OR-gate module (OR_0). Wires A and B connect the input pins of the netlist to the input pins module HA_0; wire C0 connects the third input pin of the netlist to the second input pin of module HA_1; wire P connects the first output pin of HA_0 to the first input pin of HA_1; wire S0 connects the first output pin of the netlist; wires X and Y connect the second output pins of HA_0 and HA_1 to the input pins of OR_0; finally, wire C1 connects the output pin of OR_0 to the second output pin of the netlist. Modules HA_1 and HA_0 are defined as instances of the library half adder HA1.

The module in Figure 4.3(b) can be instantiated by this component implementation. This relationship is shown graphically in Figure 4.3(d). The exterior box represents the adder module, and the interior dashed box represents the netlist implementation. Dots represent pins and lines connecting dots represent wires.

As depicted in Figure 4.3(d), the pins of a module correspond directly to the ports of the component specification that defines the module. The number of input and output pins of a module will always be equal to the cumulative widths of the input and output ports, respectively, of the specification that defines the module. Likewise, the pins of a netlist, which describes one level of decomposition of a component specification, also correspond directly to the ports of the specification. The end result is a one-to-one correspondence between the pins of a module and the pins of any netlist that implements the module's specification.

In this and other examples, component specifications and component implementations are labeled, and the labels are used to refer back to the identified structure. A component specification is labeled by its type, followed by a dot, followed by a positive integer i, which denotes that it is the ith+1 specification of that type. Thus, the three component specifications in Figure 4.3(a) are labeled ADD.0, ADD.1, and OR.0, respectively. Identical component specifications have the same label. A component implementation follows two alternative labeling schemes. If a component implementation represents a library cell, then it is labeled by the name of the cell, e.g., the half adder and 2-input OR cell specifications are labeled HA1 and OR2, respectively. If the component implementation represents a netlist, then it is labeled by the name of its corresponding component specification, followed by a dot, followed by a positive integer j, which denotes that it is the jth+1 implementation of that specification. Thus, the netlist implementation of the 1-bit full adder is labeled ADD.1.0 because it is the first implementation of specification ADD.1.

4.2.4 Decomposition Methods

A decomposition method is represented with a triple <*head*, *test*, *body*>. A method denotes a procedure for configuring a netlist of modules. The *head* of a

```
<<ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[<STYLE,?sty>]>,
 [ ?sty == HALF-ADDER ],
 [
  CONNECT-SRC(A, NETLIST, I0, 0),
  CONNECT-SRC(B, NETLIST, I1, 0),
  CONNECT-SRC(C0, NETLIST, CIN, 0),
  CONNECT-SNK(S, NETLIST, O0, 0),
  CONNECT-SNK(C1, NETLIST, COUT, 0),

  ADD-CELL(HA1, <ADD,[<I0,1>,<I1,1>],[<O0,1>,<COUT,1>],[ ]>),
  ADD-CSPEC(OR, <OR,[<I0,1>,<I1,1>],[<O0,1>],[ ]>),

  ADD-MODULE(HA_0, HA1),
  ADD-MODULE(OR_0, OR),
  ADD-MODULE(HA_1, HA1),

  CONNECT-SNK(A, HA_0, I0, 0),
  CONNECT-SNK(B, HA_0, I1, 0),
  CONNECT-SRC(P, HA_0, O0, 0),
  CONNECT-SRC(X, HA_0, COUT, 0),
  CONNECT-SNK(P, HA_1, I0, 0),
  CONNECT-SNK(C0, HA_1, I1, 0),
  CONNECT-SRC(S, HA_1, O0, 0),
  CONNECT-SRC(Y, HA_1, COUT, 0),
  CONNECT-SNK(X, OR_0, I0, 0),
  CONNECT-SNK(Y, OR_0, I1, 0),
  CONNECT-SRC,(C1, OR_0, O0, 0)
 ]>
```

Figure 4.4: Example decomposition method for 1-bit full adder.

method is a component specification *pattern*, in which port widths and attribute values can be variables as well as literal values. A variable is represented by a symbol prefixed with a question mark, e.g., ?n. As a pattern, the head of a method is *matched* against component specifications. Variables in the head are *bound* to corresponding literal values from the specification. The *test* of a method is a condition that constrains the values bound to the method's variables. The *body* of a method is a list of actions for constructing a netlist of connected subcomponents.

If the head of a method matches a component specification and its test evaluates to true, then the method is said to be *applicable* to the specification. An applicable method can be *expanded.* Two things happen when a method is expanded. First, a netlist is generated. The number of input pins and output pins of the netlist will equal the cumulative widths of the input ports and output ports of the specification, respectively. Second, the actions of the body are executed sequentially. The basic actions create component specifications, define modules, and connect pins to wires. There are also assignment statements, conditional actions, and iterative actions. The result of expanding a method is a netlist implementing one level of decomposition of the component specification under design. The modules of the netlist are defined by other component specifications but each is uninstantiated.

An example decomposition method is shown in Figure 4.4. The head of this method contains one variable (`?sty`) and matches the component specification for a 1-bit full adder (`ADD.1`) seen in Figure 4.3(a), binding `?sty` to the symbol `HALF-ADDER`. Its *test* field contains a simple condition on the value of `?sty`, which is satisfied, so the method is applicable to the specification. When expanded, this method will generate the netlist of component implementation `ADD.1.0` seen in Figure 4.3(c). The actions in the body of the method and the semantics of method expansion are described below.

4.3 Expressions, Conditions, and Actions

The test of a decomposition method consists of *conditions* and *expressions*; its body consists of *actions*, *conditions*, and *expressions*. Conditions and expressions are *evaluable*, i.e., they return a value; actions are *executable*, i.e., they produce side effects. The textual format of conditions, expressions, and actions is outlined in Figure 4.5. Their semantics are defined below.

4.3.1 Expressions

Expressions appear as arguments in conditions and actions; they can also be nested as arguments to other expressions. When an expression is encountered during the evaluation of a condition or expression or during the execution of an action, the expression is evaluated and replaced by its value. Numbers, symbols, and lists evaluate to themselves. Variables evaluate to the value to which they are *bound*. (Variables are bound when matching a component specification to the head of the method in which the variable appears; variables are also bound by BIND and LOOP actions.) It is an error to evaluate a variable that is not bound.

All operators evaluate their arguments from left to right. There are five infix arithmetic operators: `+` (addition), `-` (subtraction or unary minus), `*` (multiplication), `/` (division), and `^` (exponentiation), and four prefix operators: `FLOOR`, `CEILING`, `LOG`, and `MOD`, which have the obvious meaning. There are also three operations on lists: `LENGTH(`l`)`, which returns the number of top-level elements in list l; `INTERSECTION(`l_1`,`l_2`)`, which returns a list of elements common to both lists l_1 and l_2; and `UNION(`l_1`,`l_2`)`, which returns a list containing all the elements in lists l_1 and l_2, discarding duplicates. The operation `CONCAT({` *symbol* | *number* `}`$^+$`)` returns a symbol whose name is a concatenation of the symbols and numbers in its arguments, e.g., `CONCAT(ADD_,1,_,3)` evaluates to the symbol `ADD_1_3`.

Expressions	Conditions
number	[*cond*]
symbol	*cond* && *cond*
list	*cond* || *cond*
variable	*expr* == *expr*
[*expr*]	*expr* != *expr*
expr + *expr*	*expr* > *expr*
expr - *expr*	*expr* < *expr*
expr * *expr*	*expr* >= *expr*
expr / *expr*	*expr* <= *expr*
expr ^ *expr*	INTEGERP(x)
- *expr*	NUMBERP(x)
MOD(n, m)	SYMBOLP(x)
LOG(n, b)	LISTP(x)
FLOOR(n)	EMPTYP(x)
CEILING(n)	MEMBER(x, l)
LENGTH(l)	SUBSETP(l_1, l_2)
INTERSECTION(l_1, l_2)	INTERSECTP(l_1, l_2)
UNION(l_1, l_2)	
CONCAT({ *symbol* | *number* }$^+$)	

Actions

[*action**]
BIND(*var, value*)
CASE([<*cond, action*>*])
LOOP([*iterator**], *action*)
  iterator ::= { STEP(*var, init, limit, step*) | IN(*var, list*) }*
ADD-CSPEC(*cname, cspec*)
ADD-CELL(*cname, cspec*)
ADD-MODULE(*mname, cname*)
CONNECT-SRC(*wname*,{ NETLIST | *mname* }, *pname, i*)

Figure 4.5: Expressions, conditions, and actions.

4.3.2 Conditions

Conditions appear in the test of a method and in CASE actions. Conditions can be combined conjunctively with the && operator and disjunctively with the || operator. There are six relational operators: == (equals), != (not equals), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to). The operators == and != can be used for comparing symbols and lists as well as numbers.

There are four type predicates: INTEGERP, NUMBERP, SYMBOLP, and LISTP, which have the obvious meaning. There are also four predicates on lists: EMPTYP(x), which succeeds when x is a list of zero elements; MEMBER(x, l), which succeeds if x is a top-level element of list l; SUBSETP(l_1, l_2), which succeeds if list l_1 is a subset of list l_2; and INTERSECTP(l_1, l_2), which succeeds if one or more top-level elements of list l_1 are also top-level elements of l_2. All conditional operators evaluate their arguments from left to right. The operator && returns false on its first false argument, while || returns true on its first true argument.

4.3.3 Actions

Actions appear in the body of a method. Actions have side effects, the most important of which being to *specify* components of the netlist, to *add* modules to the netlist, and to *connect* pins of the netlist and its modules by wires. Action blocks can be created by grouping actions between square brackets. Actions in a block are executed sequentially, from left to right.

The BIND action is an assignment statement: `BIND(`*var*`,`*value*`)` evaluates its second argument and binds the variable *var* to the resulting *value*. The CASE action is an if-then-else statement: `CASE([ <`*cond*`,`*action*`>* ])` evaluates the *cond* part of each condition/action pair and executes the *action* part of the first pair whose *cond* succeeds. The LOOP action is an iterative statement: `LOOP([ `*iterator*`* ],`*action*`)` executes *action* on each iteration of the *iterator* clauses.

There are two types of iterator clauses. The first has the form

`STEP(`*var*`,`*init*`,`*limit*`,`*step*`)`.

The semantics of this clause is to bind *var* to the value of *init* and successively increment *var* by the value of *step*. For $step > 0$, iteration stops when the value bound to *var* is greater than the value of *limit*; for $step < 0$, iteration stops when *var* is less than *limit*. The second iterator type has the form

`IN(`*var*`,`*list*`)`.

Its semantics are to bind *var* to successive top-level elements of *list*. Iteration stops when the elements of *list* have been exhausted. When a loop contains multiple iterator clauses, the clauses are initialized and stepped in parallel. Iteration stops for the entire loop when iteration stops for any of the loop's iterator clauses.

The remaining actions specify modules and connect pins and wires. The action `ADD-CSPEC(`*cname*`,`*cspec*`)` inserts a named component specification, i.e., the pair `<`*cname*`,`*cspec*`>`, into the *cspecs* field of the netlist under construction. Likewise, the action `ADD-CELL(`*cname*`,`*cspec*`)` inserts a named component specification into the *cells* field of the netlist, with the exception that *cname* must identify a library cell, the specification of which equals *cspec*. The ADD-CELL action is primarily used for defining construction methods.

Likewise, the action `ADD-MODULE(`*mname*`,`*cname*`)` inserts a named module to the netlist's *modules* field, where the module is defined by the component specification identified by *cname*. The ADD-MODULE action constructs the module from the identified component specification; the input and output pins of the module will correspond to the the input and output ports of the component specification. If the component specification defining the module comes from the *cells* field of the netlist, then the *cimpl* field of the module will be instantiated by the named library cell; if

the component specification comes from the netlist's *cspecs* field, then the *cimpl* field of the module will be empty. For example, in the method shown in Figure 4.4, the action `ADD-MODULE(HA_0,HA1)` adds the pair

```
<HA_0,<[p1_00,p1_01],[p1_02,p1_03],ADD.0,HA1>>
```

to the *modules* field of the netlist in Figure 4.3(c). The *cimpl* field of this module is instantiated to the library cell `HA1`, because its component specification came from the *cells* field of the netlist.

The CONNECT-SRC and CONNECT-SNK actions assign a pin to the source and sink of a wire, respectively. Both actions take an argument list of the form

(wname, { `NETLIST` | *mname* }, *pname, i).*

The first argument, *wname*, is used to identify a wire of the netlist; *wname* must evaluate to a symbol. If no such wire exists, one is created and added to the wires of the netlist. The remaining arguments are used to identify a pin of the netlist or of one of its modules. If the second argument evaluates to the symbol `NETLIST`, then the pin will be selected from the input or output pins of the netlist under construction. Otherwise, the second argument must evaluate to a symbol (*mname*) identifying a module of the netlist; the pin will be selected from the input or output pins of that module. CONNECT-SRC and CONNECT-SNK operate by accessing the component specification associated with the second argument (i.e., the netlist under construction or one of its modules) and returning the *i*th pin corresponding to the port identified by the symbol *pname.*

As an example of how pin identification operates, consider again the decomposition method shown in Figure 4.4. The third action of this method is

```
CONNECT-SRC(CO, NETLIST, CIN, 0).
```

First, this action creates a wire `CO` and adds it to the wires of the netlist. Second, it accesses the component specification to which the method is being applied, in this case, the specification `ADD.1` from Figure 4.3(a). Third, it accesses the 0th pin corresponding to port `CIN` of the identified specification. (Because this is a CONNECT-SRC action, the pin will come from the input pins of the netlist. The first input pin corresponds to port `I0` and the second to port `I1`, so the third input pin of the netlist corresponds to the 0th pin of port `CIN` or `p0_02`.) Finally, this pin is assigned to the *src* field of the wire.

4.4 The Algorithm

I factor my definition of the component decomposition algorithm between eight functions. Of these, the first four functions, EXPAND-NETLIST, EXPAND-CSPEC,

```
EXPAND-NETLIST(netlist, filter)
  netlists = [netlist];
  for ∀ ⟨name, cspec⟩ ∈ netlist.cspecs do
    bound = [ ];
    for ∀ netlist ∈ netlists do
      for ∀ cimpl ∈ EXPAND-CSPEC(cspec, filter) do
        netlist = BIND-MODULES(cspec, cimpl, netlist);
        ADD netlist to bound;
      endfor
    endfor
    netlists = bound;
  endfor
  RETURN(netlists);
```

Figure 4.6: Definition of EXPAND-NETLIST.

EXPAND-METHOD, and BIND-MODULES, perform functional decomposition and technology mapping, while the second four functions, MATCH-METHODS, UNIFY-PORTS, UNIFY-ATTRS, and UNIFY, perform pattern matching and unification. In the definitions of these functions, lists are delimited with square brackets and tuples with angle brackets.

The top-most function, EXPAND-NETLIST, is defined in Figure 4.6. Its inputs are a netlist of uninstantiated modules and a user-defined performance filtering function. (The filtering function is used to constrain the sized of the design space and is explained later in this chapter.) The output of EXPAND-NETLIST is a list of fully-mapped netlists. Each output netlist represents an *alternative implementation* of the input netlist. Each netlist contains copies of the modules in the input netlist, where the modules are connected in the same manner and defined by the same component specifications. The difference between alternatives is that the modules of each netlist will be instantiated by distinct combinations of component implementations. Thus, each output netlist is a unique implementation of the input netlist.

The function EXPAND-NETLIST operates in the manner described below. The variable *netlists* is a list of partially mapped alternatives, which is initialized to a list containing the uninstantiated input netlist. For each component specification (*cspec*) in the *cspecs* field of the input netlist, *netlists* is modified such that uninstantiated modules defined by *cspec* are instantiated to alternative implementations (*cimpl*) of *cspec*. These implementations are generated by EXPAND-CSPEC. Module instantiation is performed by the function BIND-MODULES.

For each *netlist* in *netlists* and each *cimpl* of *cspec*, BIND-MODULES returns a copy of *netlist* in which uninstantiated modules defined by *cspec* are also copied and

```
EXPAND-CSPEC(cspec, filter)
  cimpls = [ ];
  for ∀ cell ∈ CELL-LIBRARY do
    if cell.cspec = cspec then
      ADD cell to cimpls;
    endif
  endfor
  netlists = [ ];
  for ∀ method ∈ METHOD-LIBRARY do
    bdgs = MATCH-METHOD(method, cspec);
    if MATCH-METHOD succeeds then
      netlist = EXPAND-METHOD(method, cspec, bdgs);
      ADD netlist to netlists;
    endif
  endfor
  for ∀ netlist ∈ netlists do
    for ∀ netlist ∈ EXPAND-NETLIST(netlist, filter) do
      cimpl = ⟨cspec, netlist⟩;
      ADD cimpl to cimpls;
    endfor
  endfor
  APPLY filter to cimpls;
  cimpls = cimpls returned by filter;
  RETURN(cimpls);
```

Figure 4.7: Definition of EXPAND-CSPEC.

the copies are instantiated to *cimpl*. After the last specification has been processed, *netlists* will be is returned as a list of fully-mapped alternatives of the input netlist.

The function EXPAND-CSPEC, defined in Figure 4.7, takes as its inputs the component specification to be implemented (*cspec*) and the user's filtering function. The output is a list of alternative component implementations for the input specification. The variable *cimpls* is used to collect the fully-mapped alternatives of the input specification. Technology mapping is performed by the first loop. This loop examines the cells of the given ASIC library (CELL-LIBRARY). Each library cell is represented by a component implementation, and a cell's functionality is defined by the component specification in the *cspec* field of the implementation. Cells whose specification is identical to the input specification are added to the list of implementations *cimpls*.

The second loop of EXPAND-CSPEC initiates functional decomposition. The variable *netlists* is used to collect alternative decompositions of *cspec*, which are generated by expanding applicable decomposition methods. Each method in METHOD-LIBRARY is matched against the input specification using MATCH-METHOD. Each matching (applicable) method is expanded with EXPAND-METHOD.

```
EXPAND-METHOD(method, cspec, bdgs)
  INITIALIZE netlist to ⟨[ ], [ ], [ ], [ ], [ ]⟩;
  for ∀ port ∈ cspec.iports do
    repeat port.width times do
      pin = MAKE-PIN();
      ADD pin to netlist.ipins;
    endrepeat
  endfor
  for ∀ port ∈ cspec.oports do
    repeat port.width times do
      pin = MAKE-PIN();
      ADD pin to netlist.opins;
    endrepeat
  endfor
  EXECUTE(method.body, cspec, netlist, bdgs);
  RETURN(netlist);
```

Figure 4.8: Definition of EXPAND-METHOD.

When a method is found to be applicable, MATCH-METHOD returns a binding list of pairs <*variable*, *value*>, where each *variable* comes from the head of the method and each *value* is a corresponding literal value from the input specification. For example, given the component specification

```
<ADD,[<I0,16>,<I1,16>,<CIN,1>],[<O0,16>,<COUT,1>],[<STYLE,CLA>,<LEVELS,2>]>
```

and an applicable decomposition method whose head has the form

```
<ADD,[<I0,?n>,<I1,?n>,<CIN,1>],[<O0,?n>,<COUT,1>],[<STYLE,CLA>,<LEVELS,?l>]> ,
```

MATCH-METHOD will return the binding list `[<?n,16>,<?l,2>]`. When no match is possible, MATCH-METHOD returns a failure condition.

The third loop of EXPAND-CSPEC completes functional decomposition. For each applicable method, EXPAND-METHOD returns a netlist of uninstantiated modules, with the exception of modules mapped to directly library cells as a result of the ADD-CELL action. Once all applicable methods have been expanded, each of the resulting netlists is transformed into a set of fully-mapped netlists by recursive calls to EXPAND-NETLIST. A component implementation is created for each fully-mapped netlist, and the implementation is added to *cimpls*. Finally, EXPAND-CSPEC applies the user's performance filtering function *filter* to the alternative implementations and returns the implementations accepted (returned by) the filtering function.

The function EXPAND-METHOD, defined in Figure 4.8, has three inputs: the decomposition method being expanded, the component specification being implemented, and the variable binding list returned by MATCH-METHOD. The output is

```
BIND-MODULES(cspec, cimpl, netlist)
  netlist = COPY(netlist);
  modules = [ ];
  for ∀ ⟨name, module⟩ ∈ netlist.modules do
    if cspec = module.cspec and module is uninstiated then
      module = COPY(module);
      module.cimpl = cimpl;
    endif
    ADD ⟨name, module⟩ to modules;
  endfor
  netlist.modules = modules;
  RETURN(netlist);
```

Figure 4.9: Definition of BIND-MODULES.

a netlist of uninstantiated modules. This netlist represents one level of decomposition implementing the component specification. The input and output pins of the netlist are created to correspond to the ports of the component specification. The modules of the netlist and their wire connections are generated by executing the actions in the body of the method. The semantics of action execution were described earlier in Section 4.3.3.

The function BIND-MODULES, defined in Figure 4.9, instantiates the modules of a netlist to an implementation of the specification defining the modules. This function takes as its input a component specification (*cspec*), an implementation of that specification (*cimpl*), and a partially mapped netlist. One or more of the netlist's uninstantiated modules will be defined by *cspec*. The output of BIND-MODULES is a copy of the netlist in which copies of these modules are instantiated by *cimpl*.

The function MATCH-METHOD, defined in Figure 4.10, is used to determine if a method is applicable to a component specification. When applicable, MATCH-METHOD outputs a list of variable bindings collected during the matching (or *unification*) process. Returning an empty binding list does not denote failure. Bindings are collected in the variable *bdgs*, which is initialized to the empty list. The head of a method (*cptrn*) matches the input specification (*cspec*) if they have the same function type and if their ports and attributes *unify*. The method is applicable if its head and the input specification match and if the method's test evaluates successfully.

The functions UNIFY-PORTS and UNIFY-ATTRS compare the ports and attributes of a component specification against the port and attribute patterns in the head of a method. I refer to the ports and attributes of a method's head as patterns, because they may contain variables in the place of literal values. A port and a port

```
MATCH-METHOD(method, cspec)
  bdgs = [ ];
  cptrn = method.head;
  if cspec.type ≠ cptrn.type then FAIL; endif
  bdgs = UNIFY-PORTS(cspec.iports, cptrn.iports, bdgs);
  if UNIFY-PORTS fails then FAIL; endif
  bdgs = UNIFY-PORTS(cspec.oports, cptrn.oports, bdgs);
  if UNIFY-PORTS fails then FAIL; endif
  bdgs = UNIFY-ATTRS(cspec.attrs, cptrn.attrs, bdgs);
  if UNIFY-ATTRS fails then FAIL; endif
  if EVAL(method.test, bdgs) fails then FAIL; endif
  SUCCESS: RETURN(bdgs);
```

Figure 4.10: Definition of MATCH-METHODS.

pattern unify if their names are the same and their widths unify. Likewise, an attribute and an attribute pattern unify if their names are the same and their values unify.

Finally, the function UNIFY tests if two terms x and y unify, where y is possibly a variable. If y is a variable, then the two terms unify if y is not in already bound in the given binding list *bdgs*, which results in a binding of y to x being added, or if the value bound to y equals x. If y is not a variable, than the two values unify if x and y are equal. UNIFY returns the resulting binding list, modified or not.

4.5 Controlling Search

The component decomposition algorithm performs search through a hierarchical design space of component decompositions. The essence of the algorithm is to find all possible decompositions of the modules in the input netlist and to return all possible combinations of those decompositions as the alternative implementations of the netlist. If otherwise unconstrained, the size of the design space would make this algorithm computationally intractable. In particular, the size of the design space would be proportional to the product of the number of alternative implementations of each module in a given netlist. Even for small components, such as a 16-bit adder, there could be several hundred thousand alternative implementations, only a handful of which make sense to generate. To focus in on the "reasonable" implementations and control the amount of search, the component decomposition algorithm is defined using a branch-and-bound approach that acts to constrain the size of the design space.

```
UNIFY-PORTS(ports, ptrns, bdgs)
  if LENGTH(ports) ≠ LENGTH(ptrns) then FAIL; endif
  for ∀ port ∈ ports and ∀ ptrn ∈ ptrns do
    if port.name ≠ ptrn.name then FAIL; endif
    bdgs = UNIFY(port.width, ptrn.width, bdgs);
    if UNIFY fails then FAIL; endif
  endfor
  SUCCESS: RETURN(bdgs);

UNIFY-ATTRS(attrs, ptrns, bdgs)
  if LENGTH(attrs) ≠ LENGTH(ptrns) then FAIL; endif
  for ∀ attr ∈ attrs and ∀ ptrn ∈ ptrns do
    if attr.name ≠ ptrn.name then FAIL; endif
    bdgs = UNIFY(attr.value, ptrn.value, bdgs);
    if UNIFY fails then FAIL; endif
  endfor
  SUCCESS: RETURN(bdgs);

UNIFY(x, y, bdgs)
  if y is a variable then
    if ∃ ⟨var, value⟩ ∈ bdgs s.t. y = var then
      if value ≠ x then FAIL; endif
    else
      add ⟨y, x⟩ to bdgs;
    endif
  elseif x ≠ y then FAIL; endif
  SUCCESS: RETURN(bdgs);
```

Figure 4.11: Definition of UNIFY-PORTS, UNIFY-ATTRS, and UNIFY.

Two controlling principles are used to constrain design space expansion. The first principle is to ignore netlist implementations containing two or more modules defined by the same component specification that are not instances of the same component implementation. The second principle is to apply the user-defined preformance filtering function to lists of competing alternative subcomponent implementations, discarding all implementations not accepted by the filtering function. The first principle is captured in the function BIND-MODULES; the second principle is captured in the function EXPAND-CSPEC. Together, these two principle significantly reduce the size of the design space, while allowing the user to focus on those implementations that best characterize his design requirements.

As an example of the first principle, consider the component implementation and netlist for a 1-bit full adder shown in Figure 4.3(c). The netlist contains two half adder modules, `HA_0` and `HA_1`, both of which are defined by the same component specification and mapped directly to a library cell, and a 2-input OR-gate. Suppose

```
SELECT-PERCENTAGE-SMALLEST(cspec, cimpls)
  SORT cimpls by increasing area;
  smallest = cimpls.first;
  best = [smallest];
  max = AREA(smallest) × (1 + FILTER-EPSILON);
  for ∀ cimpl ∈ cimpls.rest s.t. AREA(cimpl) > max do
    if MAXDELAY(cimpl) < MAXDELAY(smallest) then
      ADD cimpl to best;
      smallest = cimpl;
    endif
  endfor
  RETURN(best);
```

Figure 4.12: Filtering function to select percentage of smallest implementations.

that instead mapping the half adder modules to library cells they are actually mapped to three alternative implementations of a half adder, then there are 3^2 possible implementations of the netlist; a netlist requiring four 1-bit adders would consequently have $(3^2)^4$ or 4096 possible implementations. By applying the first principle, the component decomposition algorithm will only generate three implementations of the former netlist and, likewise, only three implementations of the latter. Although this principle will ignore a small percentage of "desirable" designs obtained by mixing module instantiations, it is consistent with the practices of human designers. In addition, it is possible to imagine a postprocess optimizer that could detect these designs.

The user-defined performance filtering function provides a mechanism for evaluating and discarding competing alternative implementations of the same subcomponents. Because the implementations are fully mapped, the area, delay, and power cost estimates can be computed for each. These estimates can be used to compare and rank alternative implementations. Possible filtering strategies are to discard all but the smallest implementation or all but the fastest or cheapest. Another strategy is to discard all implementations that are not within some percentage distance from the smallest, fastest, or cheapest, or to discard implementations that do not make favorable trade-offs between area, delay, and cost. A filtering function can also be used as an interface point for designer interaction, i.e., allowing a designer to examine the alternatives and make his own choices.

The filtering function is expected to take two inputs. The first input is the component specification being implemented, and the second is the list of alternative implementations of that specification. The filter function is expected return a list containing only those implementations whose performance characteristics are favored by the user. Two example filtering functions are shown in Figures 4.12 and 4.13.

```
SELECT-BOUNDED-CURVE(cspec, cimpls)
  SORT cimpls by increasing area and increasing delay;
  fastest = smallest = cimpls.first;
  min = MAXDELAY(fastest);
  for ∀ cimpl ∈ cimpls do
    if MAXDELAY(cimpl) < min then
      fastest = cimpl;
      min = MAXDELAY(cimpl);
    endif
  endfor
  if smallest = fastest then RETURN([smallest]); endif
  best = [smallest, fastest];
  last-m = (AREA(smallest)−AREA(fastest))÷(MAXDELAY(smallest)−MAXDELAY(fastest));
  for ∀ cimpl ∈ cimpls.rest until cimpl = fastest do
    m = (AREA(cimpl)−AREA(fastest))÷(MAXDELAY(cimpl)−MAXDELAY(fastest));
    if |m| < |last-m| then
      ADD cimpl to best;
      last-m = m;
    endif
  endfor
  RETURN(best);
```

Figure 4.13: Filtering function to selects increasingly favorable implementations.

The filtering function SELECT-PERCENTAGE-SMALLEST, which is defined in Figure 4.12, discards all implementations except those that are within a percentage distance (FILTER-EPSILON) of the implementation with the least area (*smallest*). This function first sorts the competing implementations (*cimpls*) by area, from smallest to largest. The variable *max* is set to the maximum area of acceptable implementations. All implementations that have an area no greater than *max* and that are increasingly faster are collected into the variable *best*. Upon encountering the first implementation whose area is greater than *max*, the list acceptable implementations collected in *best* are returned.

The filtering function SELECT-BOUNDED-CURVE, defined in Figure 4.13, discards all implementations that do not make "desirable" trade-offs between area and delay. Competing implementations are again sorted by increasing area; implementations with equal area are ordered by increasing delay. The function then identifies the implementation with the least area (*smallest*) and the implementation with the least delay (*fastest*), where the function MAXDELAY computes the maximum delay through a fully mapped implementation. If the smallest implementation is also the fastest, then only that one implementation is returned as acceptable. Otherwise, all implementations from *smallest* to *fastest* that make increasingly-favorable trade-offs between area and delay are collected and returned. An implementation is considered

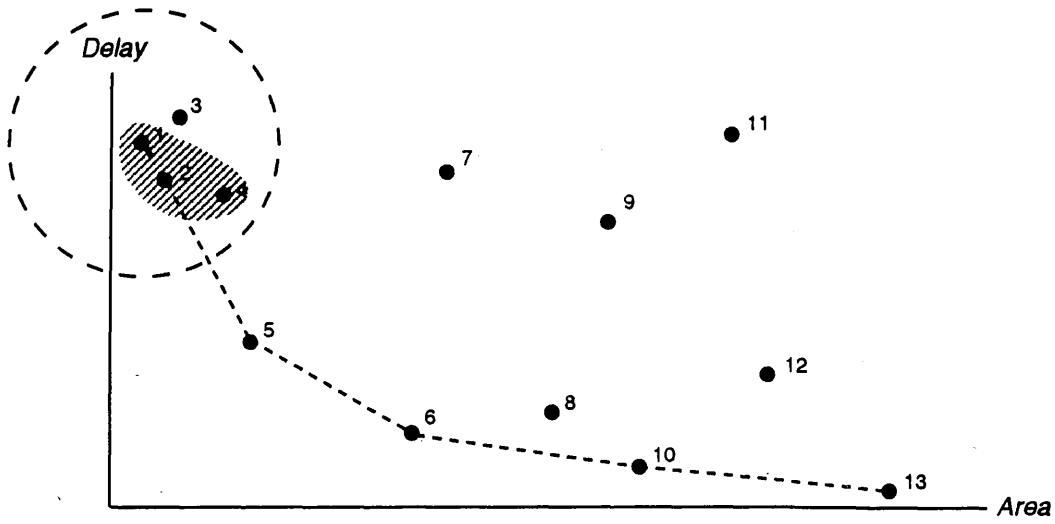


Figure 4.14: Example delay *vs* area design space.

to be "increasingly favorable" if the ratio ($|m|$) of the difference between the implementation's delay and area and the delay and area of *fastest* is less than that ratio ($|$*last-m*$|$) for the last favorable implementation and *fastest*.

An example design space is shown in Figure 4.14. This space contains 13 alternative implementations, which appear as block dots labeled in sorted order. The leftmost implementation (1) has the least area; the rightmost (13) has the least delay. Given that the dashed circle surrounds the space of designs within FILTER-EPSILON of the smallest implementation, the filtering function SELECT-PERCENTAGE-SMALLEST would return the three implementations contained in the shaded region as acceptable. Another implementation (3), also within FILTER-EPSILON of the smallest is not returned as acceptable because implementations 1 and 2 are both smaller and faster. For SELECT-BOUNDED-CURVE, a dashed line connects the six implementations from smallest to fastest, inclusive, that would be returned as making increasingly-favorable trade-offs between area and delay. Although implementations 4 and 8 make favorable trade-offs between the smallest and fastest, they do not make favorable trade-offs between implementations 2 and 6, respectively.

Chapter 5
Component Decomposition Examples

In this chapter, I step through two applications of the component decomposition algorithm to further clarify concepts discussed in Chapter 4. First, I present an informal explanation of these examples; then, I demonstrate how the algorithm decomposes the specification of a 4-bit adder into a ripple-carry implementation; finally, I demonstrate how alternative implementations are generated. Throughout both examples, assume a simplistic performance filtering function that returns all input implementations.

5.1 Overview

There are two aspects of the component decomposition algorithm that I wish to demonstrate through examples. The first is the use of the algorithm for decomposition and technology mapping. The second is its use in exploring the space of design alternatives. Examples are presented in detail, so in this section, I provide a brief explanation of each.

In Section 5.2, I present an example that decomposes a 4-bit ripple-carry adder, mapping it into library half adder cells and 2-input OR gates. The example input netlist is shown in Figure 5.1. This netlist contains one module defined by a component specification for a 4-bit adder; the module is uninstantiated. An applicable decomposition method is shown in Figure 5.4. This method matches the input specification, binding variable `?n` to 4. The actions in its body first connect wires to the input and output pins of the netlist, then iteratively define `?n` 1-bit adder modules, connecting wires to the pins of each. The resulting netlist generated by expanding this method is shown in Figure 5.5.

The netlist in Figure 5.5 contains four uninstantiated 1-bit adder modules defined by the same component specification. An applicable decomposition method is

shown in Figure 5.3. The actions in the body of this method first connect wires to the input and output pins of the netlist, then define two half adder modules and an OR gate modules, and finally connect wires to the pins of these modules. The resulting netlist generated by expanding this method is shown in Figure 5.6.

The netlist in Figure 5.6 contains three modules, two of which are defined by the specification for a half adder and the other by the specification of an OR gate. There are no applicable decomposition methods for these specifications, but, as it happens, the cell library provided for this example contains two identically specified cells (`HA` and `OR2`), as depicted in Figure 5.2. These cells can be used to instantiate the modules of the netlist in Figure 5.6, which can be used as a netlist implementation to instantiate the four adder modules of the netlist in Figure 5.5, which can be used as a netlist implementation to instantiate the 4-bit adder module in Figure 5.1. The resulting fully-mapped implementation of the input netlist is shown in Figure 5.7 and depicted graphically in Figure 5.8.

In Section 5.3, I present a slightly less detailed example that depicts how to decompose an n-bit adder into several alternative implementations that mix the ripple-carry and carry look-ahead design styles. The top-level method for adder decomposition is shown in Figure 5.4. The head of this method matches the specification for a generic n-bit adder with no attributes. Its actions map the generic adder into a ripple-carry adder with at most `?1` levels of carry look-ahead, where `?1` is bound to $n \log 4$, assuming a 4-bit carry look-ahead generator.

The two decomposition methods shown in Figures 5.10 and 5.11 reflect two alternative methods for implementing an n-bit adder with at most l levels of look-ahead. The first method implements it by rippling $\frac{n}{4^l}$ full carry look-ahead adders. The second method maps it into a ripple-carry adder with at most $l-1$ levels of carry look-ahead. Until $l = 0$, both methods will be applicable to the adder specified by the actions of the second method, resulting in the design of adders using a full carry look-ahead style to a full ripple-carry style with a mix of both styles in between.

5.2 Decomposition and Technology Mapping

In this section, I present a detailed demonstration of the decomposition and instantiation of a 4-bit ripple-carry adder. For this example, there is only one possible implementation.

5.2.1 Input Netlist

The input netlist specification for this example is shown in Figure 5.1(a) and depicted graphically in Figure 5.1(b). In this and other such figures, the dots represent individual input and output pins and the lines connecting them represent wires; wire names appear to the right. Additionally, netlists are represented by dashed boxes, and modules are represented by solid boxes.

The first two fields of the netlist contain the list of input pins

```
[p0_00,p0_01,p0_02,p0_03,p0_04,p0_05,p0_06,p0_07,p0_08]
```

and output pins

```
[p0_09,p0_10,p0_11,p0_12,p0_13].
```

Pin names must be distinct within a netlist and its modules but not across netlists or in embedded netlists that describe the implementations of modules. In this example, I reuse pin names such as `p0_00` in several netlists.

The third field of the netlist contains a list of wires, which specify how pins are connected. The wire

```
<A0,p0_00,[p1_00]>
```

specifies that pin `p0_00` is connected to pin `p1_00`. Pin `p0_00` is the source of the wire; pin `p1_00` is its single sink. Wires are identified by their name; the name of the wire seen above is `A0`.

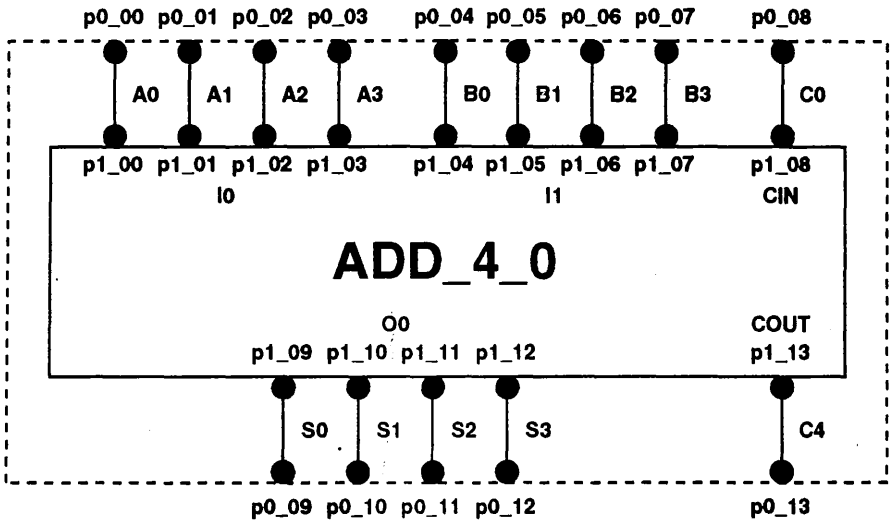
Following the list of wires is the list of named component specifications and the list of named modules. This netlist consists of a single module, the functionality of which is defined by the component specification

```
<ADD,[<I0,4>,<I1,4>,<CIN,1>],[<O0,4>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,0>]>.
```

This specification designates that the module is a 4-bit full adder. It has three input ports, `I0`, `I1`, and `CIN`, and two output ports, `O0` and `COUT`. The ports `I0`, `I1`, and `O0` have a width of four; the ports `CIN` and `COUT` have a width of one. The specification also contains two attributes. The first is used to indicate that the adder is to be implemented using a ripple-carry style, and the second to indicate that 0 levels of look ahead are to be used.

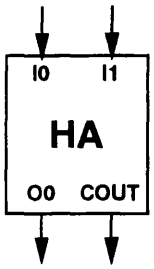
```
< [p0_00,p0_01,p0_02,p0_03,p0_04,p0_05,p0_06,p0_07,p0_08],
 [p0_09,p0_10,p0_11,p0_12,p0_13],
 [  <A0,p0_00,[p1_00]>,<A1,p0_01,[p1_01]>,<A2,p0_02,[p1_02]>,<A3,p0_03,[p1_03]>,
    <B0,p0_04,[p1_04]>,<B1,p0_05,[p1_05]>,<B2,p0_06,[p1_06]>,<B3,p0_07,[p1_07]>,
    <C0,p0_08,[p1_08]>,<S0,p1_09,[p0_09]>,<S1,p1_10,[p0_10]>,<S2,p1_11,[p0_11]>,
    <S3,p1_12,[p0_12]>,<C4,p1_13,[p0_13]> ],
 [ ],
 [ <ADD_4,<ADD,[<I0,4>,<I1,4>,<CIN,1>],[<O0,4>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,0>]>> ],
 [ <ADD_4_0,   <[p1_00,p1_01,p1_02,p1_03,p1_04,p1_05,p1_06,p1_07,p1_08],
               [p1_09,p1_10,p1_11,p1_12,p1_13],
               <ADD,[<I0,4><I1,4>,<CIN,1>],[<O0,4>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,0>]>, >> ] >
```

(a)



(b)

Figure 5.1: Input netlist for a 4-bit adder: (a) textual form; and (b) graphical depiction.



<HA1, <ADD,[<I0,1>,<I1,1>],[<O0,1>,<COUT,1>],[ ]>, [<LOAD,...>,<AREA,...>,<DELAY,...>]>

(a)

<OR2, <OR,[<I0,2>],[<O0,1>],[ ]>, [<LOAD,...>,<AREA,...>,<DELAY,...>]>

(b)

Figure 5.2: Sample cell library: (a) half adder; and (b) 2-input OR gate.

5.2.2 Cell Library

The example cell library is shown in Figure 5.2. This is a very small and limited library consisting of a half adder (`HA`) and a 2-input OR gate (`OR2`). Each cell in the library is represented with a component implementation.

The first field of the component implementation contains a component specification, which defines the functionality of the cell. The second field is empty. The third field contains a list of properties, which define such things as the cell's name and its physical characteristics. The values for the area, delay, and cost of the cells are not shown here because they are not needed for this example.

5.2.3 Expanding The Input Netlist

When the netlist from Figure 5.1 is input to EXPAND-NETLIST, the netlist's one component specification

```
<ADD,[<IO,4>,<I1,4>,<CIN,1>],[<OO,4>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,0>]>
```

is passed to EXPAND-CSPEC. The function EXPAND-CSPEC first looks for all library cells that are identically specified; in this case, no such cells exist. Next, EXPAND-CSPEC looks for all applicable decomposition methods. A method is applicable to a component specification if its head matches the specification and if its test evaluates successfully. A method's applicability is determined by the function MATCH-METHOD; if applicable, MATCH-METHOD returns the binding list of variables unified during the matching process.

Figures 5.3(a) and 5.4(a) show the two decomposition methods required to run this example; these methods are depicted graphically in Figures 5.3(b) and 5.4(b). The head of the method in Figure 5.3(a),

```
<ADD,[<IO,1>,<I1,1>,<CIN,1>],[<OO,1>,<COUT,1>],[ ]>,
```

fails to match the component specification because its ports are the wrong width and it has no attributes. Thus, this method is not applicable.

On the other hand, the head of the method in Figure 5.4(a),

```
<ADD,[<IO,?n>,<I1,?n>,<CIN,1>],[<OO,?n>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,0>]>,
```

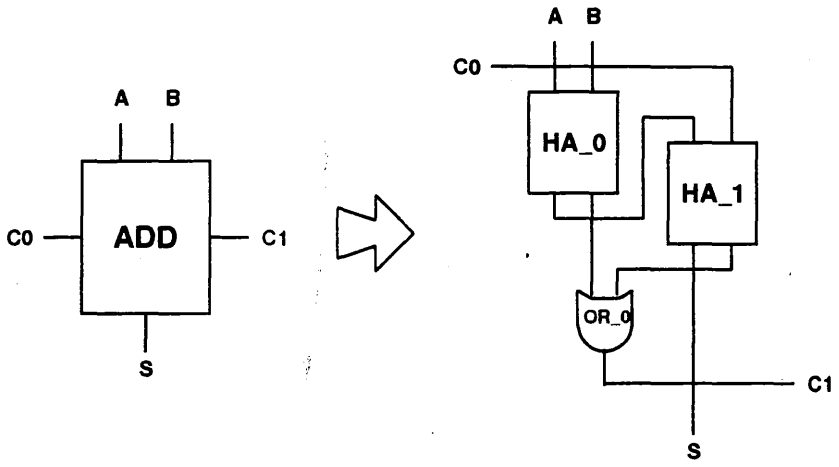
matches the component specification of the input netlist. Variable `?n` unifies consistently to 4, in which case the method's test `[ ?n > 1 ]` succeeds. Thus, the the method is applicable. The function MATCH-METHOD returns the binding list `[<?n,4>]`.

5.2.4 Method Expansion

For applicable methods, EXPAND-CSPEC calls EXPAND-METHOD, which constructs a netlist of connected modules. First, EXPAND-METHOD creates an empty netlist. Then, it creates lists of input and output pins, where the length of these lists equals the cumulative widths of the input and output ports of the component specification. Finally, it executes the method's actions.

```
<<ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[ ]>,
 [ ],
 [
   CONNECT-SRC(A, NETLIST, I0, 0),
   CONNECT-SRC(B, NETLIST, I1, 0),
   CONNECT-SRC(C0, NETLIST, CIN, 0),
   CONNECT-SNK(S, NETLIST, O0, 0),
   CONNECT-SNK(C1, NETLIST, COUT, 0),

   ADD-CELL(HA1, <ADD,[<I0,1>,<I1,1>],[<O0,1>,<COUT,1>],[ ]>),
   ADD-CSPEC(OR, <OR,[<I0,1>,<I1,1>],[<O0,1>],[ ]>),

   ADD-MODULE(HA_0, HA1),
   ADD-MODULE(HA_1, HA1),
   ADD-MODULE(OR_0, OR),

   CONNECT-SNK(A, HA_0, I0, 0),
   CONNECT-SNK(B, HA_0, I1, 0),
   CONNECT-SRC(P, HA_0, O0, 0),
   CONNECT-SRC(X, HA_0, COUT, 0),
   CONNECT-SNK(P, HA_1, I0, 0),
   CONNECT-SNK(C0, HA_1, I1, 0),
   CONNECT-SRC(S, HA_1, O0, 0),
   CONNECT-SRC(Y, HA_1, COUT, 0),
   CONNECT-SNK(X, OR_0, I0, 0),
   CONNECT-SNK(Y, OR_0, I1, 0),
   CONNECT-SRC,(C1, OR_0, O0, 0)
 ]>
```

(a)

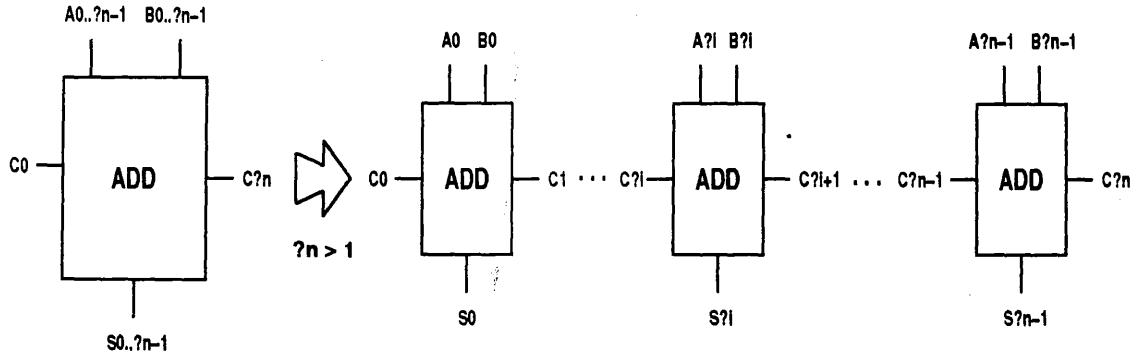


(b)

Figure 5.3: Sample decomposition method for 1-bit adder: (a) textual specification; and (b) graphical depiction.

```
<<ADD,[<I0,?n>,<I1,?n>,<CIN,1>],[<O0,?n>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,0>]>,
 [ ?n > 1 ],
 [
  LOOP( [ STEP(?i,0,?n-1,1) ],
    [
      CONNECT-SRC(CONCAT(A,?i), NETLIST, I0, ?i),
      CONNECT-SRC(CONCAT(B,?i), NETLIST, I1, ?i),
      CONNECT-SNK(CONCAT(S,?i), NETLIST, O0, ?i)
    ]),
  CONNECT-SRC(C0, NETLIST, CIN, 0),
  CONNECT-SNK(CONCAT(C,?n), NETLIST, COUT, 0),
  ADD-CSPEC(ADD-1, <ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[ ]>),
  LOOP( [ STEP(?i,0,?n-1,1) ],
    [
      ADD-MODULE(CONCAT(ADD-1_, ?i), ADD-1),
      CONNECT-SNK(CONCAT(A,?i), CONCAT(ADD-1_, ?i), I0, 0),
      CONNECT-SNK(CONCAT(B,?i), CONCAT(ADD-1_, ?i), I1, 0),
      CONNECT-SNK(CONCAT(C,?i), CONCAT(ADD-1_, ?i), CIN, 0),
      CONNECT-SRC(CONCAT(S,?i), CONCAT(ADD-1_, ?i), O0, 0),
      CONNECT-SRC(CONCAT(C,?i+1), CONCAT(ADD-1_, ?i), COUT, 0)
    ]),
 ] >
```

(a)



(b)

Figure 5.4: Sample decomposition method for ?n-bit ripple-carry adder: (a) textual specification; and (b) graphical depiction.

In this example, the input ports of the component specification have a cumulative width of nine and the output ports have a cumulative width of five, so the netlist under construction will have nine input pins,

```
[p0_00,p0_01,p0_02,p0_03,p0_04,p0_05,p0_06,p0_07,p0_08]
```

and five output pins

```
[p0_09,p0_10,p0_11,p0_12,p0_13].
```

The first four input pins correspond to port `I0` of the component specification, the next four to `I1`, and the last input pint to `CIN`. Likewise, the first four output pins correspond to port `O0` and the last output pin to `COUT`.

5.2.5 Action Execution

After creating the pins of the netlist, EXPAND-METHOD executes the actions in the body of the method. The input to the **execute** function includes the body of the method being expanded, the specification being implemented, the netlist being constructed, and the binding list. Action execution adds modules and wires to the netlist or references and accesses variables in the binding list. The component specification is used to reference and access pins of the netlist.

The body of the method from Figure 5.4(a) consists of an action block. Each action within this block is executed sequentially. The first is a LOOP action,

```
LOOP([STEP(?i, 0, ?n-1, 1)],
  [
    CONNECT-SRC(CONCAT(A, ?i), NETLIST, I0, ?i),
    CONNECT-SRC(CONCAT(B, ?i), NETLIST, I1, ?i),
    CONNECT-SNK(CONCAT(S, ?i), NETLIST, O0, ?i),
  ])
```

The variable `?i` is added to the binding list, bound to 0. The action block of the loop is executed iteratively, stepping `?i` from 0 to `?n-1` (3) by 1.

With `?i` bound to 0, the first action of the embedded action block connects a wire to an input pin of the netlist, making it the source pin of the wire. The wire's name is `A0`; since this is the first reference to a wire of that name for the given netlist, the wire is created and added to the netlist's wires. The pin is the 0th pin of the pins corresponding to the port named `I0` of the component specification being implemented. Port `I0` corresponds to the first four input pins of the netlist, the 0th pin of which is `p0_00`. The second action assigns a source pin to wire `B0`. Port `I1` corresponds to the second four input pins of the netlist, the 0th pin of which is `p0_04`. Likewise, the third action adds the output pin `p0_09` to the sinks of wire `S0`.

After executing the LOOP action, the netlist will contain the list of wires

```
[<A0,p0_00,[ ]>,<B0,p0_04,[ ]>,<S0,,[p0_09]>,
 <A1,p0_01,[ ]>,<B1,p0_05,[ ]>,<S1,,[p0_10]>,
 <A2,p0_02,[ ]>,<B2,p0_06,[ ]>,<S2,,[p0_11]>,
 <A3,p0_03,[ ]>,<B3,p0_07,[ ]>,<S3,,[p0_12]>]
```

In a similar fashion, the next two actions

```
CONNECT-SRC(C0, NETLIST, CIN, 0),
CONNECT-SNK(CONCAT(C, ?n), NETLIST, COUT, 0)
```

create two additional wires for the carry input and the carry output of the adder.

The next action

```
ADD-CSPEC(ADD_1,<ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[ ]>)
```

adds a pair consisting of the symbol ADD_1 and the component specification for a 1-bit adder,

```
<ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[ ]>
```

to the *cspecs* field of the netlist. The symbol ADD_1 is used to refer to this specification when creating modules.

Finally, the LOOP action

```
LOOP([STEP(?i, 0, ?n-1, 1)],
  [
    ADD-MODULE(CONCAT(ADD_1_, ?i), ADD_1),
    CONNECT-SNK(CONCAT(A, ?i), CONCAT(ADD_1_, ?i), I0, 0),
    CONNECT-SNK(CONCAT(B, ?i), CONCAT(ADD_1_, ?i), I1, 0),
    CONNECT-SNK(CONCAT(C, ?i), CONCAT(ADD_1_, ?i), CIN, 0),
    CONNECT-SRC(CONCAT(S, ?i), CONCAT(ADD_1_, ?i), O0, 0),
    CONNECT-SRC(CONCAT(C, ?i+1), CONCAT(ADD_1_, ?i), COUT, 0)
  ]
```

creates four 1-bit adder modules and connects their pins. When ?i is bound to 0, the ADD-MODULE action creates a module defined by the 1-bit adder specification associated with the symbol ADD_1 and pairs this module with the symbol ADD_1_0 in the netlist's *modules* field. The input and output pins of the module are created to correspond to the ports of the component specification, so there are three input pins

```
[p1_00,p1_01,p1_02]
```

and two output pins

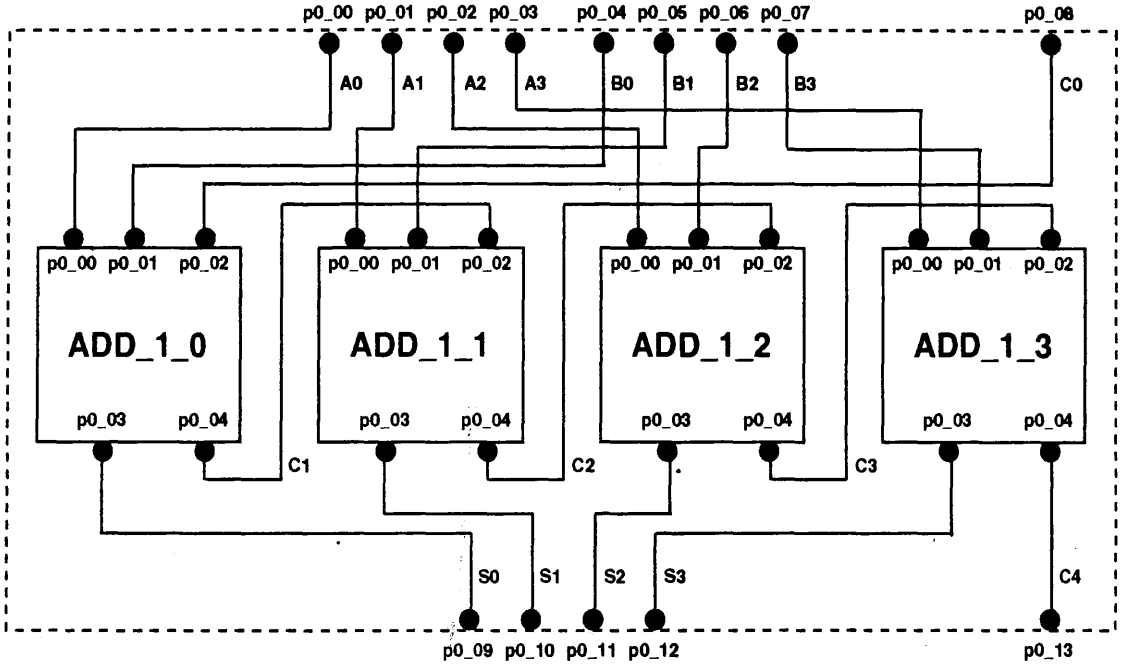
```
[p1_03,p1_04].
```

The next five actions add the module's input pins to the sinks of wires A0, B0, and C0 and assign its output pins to the sources of the wires S0 and C1, which will be connected to the carry input pin of the next module, ADD_1_1.

When completed, EXPAND-METHOD will return an uninstantiated netlist implementing a ripple-carry adder. This netlist is shown in Figure 5.5(a). This netlist is depicted graphically in Figure 5.5(b).

```
< [p0_00,p0_01,p0_02,p0_03,p0_04,p0_05,p0_06,p0_07,p0_08],
 [p0_09,p0_10,p0_11,p0_12,p0_13],
 [ <A0,p0_00,[p1_00]>,<A1,p0_01,[p2_00]>,<A2,p0_02,[p3_00]>,<A3,p0_03,[p4_00]>,
   <B0,p0_04,[p1_01]>,<B1,p0_05,[p2_01]>,<B2,p0_06,[p3_01]>,<B3,p0_07,[p4_01]>,
   <S0,p1_03,[p0_09]>,<S1,p2_03,[p0_10]>,<S2,p3_03,[p0_11]>,<S3,p4_03,[p0_12]>,
   <C0,p0_08,[p1_02]>,<C1,p1_04,[p2_02]>,<C2,p2_04,[p3_02]>,<C3,p3_04,[p4_02]>,
   <C4,p4_04,[p0_13]> ],
 [ ],
 [ <ADD_1,<ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[ ]>> ],
 [ <ADD_1_0, <[p1_00,p1_01,p1_02], [p1_03,p1_04], <ADD, [<I0,1>,<I1,1>,<CIN,1>], [<O0,1>,<COUT,1>], [ ]>, >>,
   <ADD_1_1, <[p2_00,p2_01,p2_02], [p2_03,p2_04], <ADD, [<I0,1>,<I1,1>,<CIN,1>], [<O0,1>,<COUT,1>], [ ]>, >>,
   <ADD_1_2, <[p3_00,p3_01,p3_02], [p3_03,p3_04], <ADD, [<I0,1>,<I1,1>,<CIN,1>], [<O0,1>,<COUT,1>], [ ]>, >>,
   <ADD_1_3, <[p4_00,p4_01,p4_02], [p4_03,p4_04], <ADD, [<I0,1>,<I1,1>,<CIN,1>], [<O0,1>,<COUT,1>], [ ]>, >> ] >
```

(a)



(b)

Figure 5.5: Netlist for 4-bit ripple-carry adder: (a) textual form; and (b) graphical depiction.

5.2.6 Instantiation and Mapping

This brings us back to EXPAND-CSPEC. After expanding each of the applicable methods, EXPAND-CSPEC recursively applies EXPAND-NETLIST to the resulting netlists. For this example, there is only one netlist. EXPAND-NETLIST will apply EXPAND-CSPEC to the component specification for the 1-bit full adder,

```
<ADD,[<IO,1>,<I1,1>,<CIN,1>],[<OO,1>,<COUT,1>],[ ]>.
```

Again, EXPAND-CSPEC will find no library cells that are defined by their component specification, but it will find that the method from Figure 5.3(a) is applicable. Since there are no variables in the head of this method, the binding list returned by MATCH-METHOD is empty (i.e., [ ]).

The netlist created when this method is expanded will have three input pins and two output pins. The first five actions in the body of the method,

```
CONNECT-SRC(A, NETLIST, IO, 0),
CONNECT-SRC(B, NETLIST, I1, 0),
CONNECT-SRC(CO, NETLIST, CIN, 0),
CONNECT-SNK(S, NETLIST, OO, 0),
CONNECT-SNK(C1, NETLIST, COUT, 0)
```

create the wires connected to the input and output pins of the netlist. The actions,

```
ADD-CSPEC(HA,<ADD,[<IO,1>,<I1,1>],[<OO,1>,<COUT,1>],[ ]>),
ADD-CSPEC(OR,<OR,[<IO,1>,<I1,1>],[<OO,1>],[ ]>)
```

add component specifications for a half adder (HA) and 2-input OR gate (OR) to the *cspecs* field of the netlist, while the actions

```
ADD-MODULE(HA_0, HA),
ADD-MODULE(HA_1, HA),
ADD-MODULE(OR_0, OR)
```

create two half-adder modules and one OR-gate module and adds them to the *modules* field of the netlist. The final actions connect up the wires that turn three modules into a full adder.

The resulting netlist returned by EXPAND-METHOD is shown in Figure 5.6(a); its graphical depiction appears in Figure 5.6(b). This netlist will be passed to EXPAND-NETLIST, which will, in turn. apply EXPAND-CSPEC to the component specifications for the half adder

```
<ADD,[<IO,1>,<I1,1>],[<OO,1>,<COUT,1>],[ ]>
```

and the OR gate

```
<OR,[<I0,1>,<I1,1>],[<O0,1>],[ ]>.
```

This time, EXPAND-CSPEC will find library cells that match both specifications and no applicable methods. These will be passed through the performance filter, which for this example does nothing, and returned to EXPAND-NETLIST, which will assign them to the *cimpl* fields of the modules in the netlist from Figure 5.6(a).

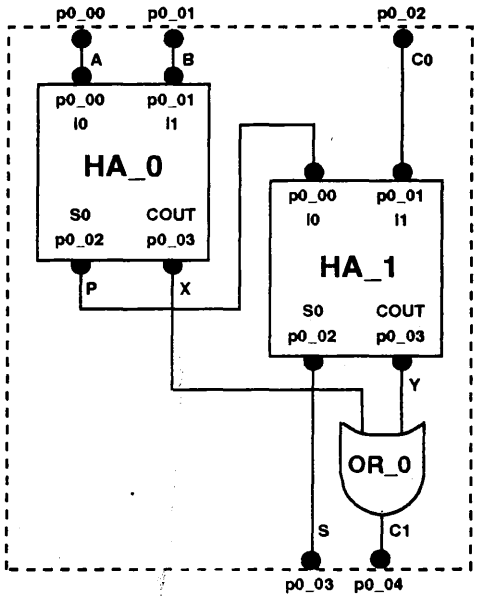
Now fully mapped to library cells, EXPAND-NETLIST returns the netlist to EXPAND-CSPEC, where it will be assigned to a component implementation and returned to the higher-level invocation of EXPAND-NETLIST. This implementation of the 1-bit adder will be assigned to the *cimpl* fields of the four 1-bit adder modules in the netlist of Figure 5.5(a) and the resulting, fully-mapped netlist will be returned to the next higher-level invocation of EXPAND-CSPEC. Again, a component implementation will be created for the netlist, and it will be passed through the performance filter and returned to the top-most invocation of EXPAND-NETLIST, where the implementation will be bound to the 4-bit adder module and the example is completed.

The final fully-mapped netlist is shown in Figure 5.7. For brevity, component implementations are labeled and their assignment to the *cimpl* fields of modules is by reference. Figures 5.7(a) and (b), OR2 and HA, show the implementations representing the 2-input OR gate and the half adder from the cell library, Figure 5.7(c), ADD1, show the implementation of the 1-bit adder, and Figure 5.7(d), ADD4, shows the implementation of the 4-bit adder. A graphical depiction of the embedded netlists appears in Figure 5.8.

There are several important aspects of the component decomposition algorithm that are illustrated by this example. First, this example shows how functional specifications are used as an alternative to Boolean graphs for the purpose of technology mapping. Using functional specification, the half adder library cell is described as succinctly as the 2-input OR gate. Second, this example shows how decomposition methods are used to encode design styles, and how they are expanded into hierarchical netlists of structural components. One final point that is illustrated by this example is the need for technology-specific decomposition methods. For instance, the decomposition method for implementing a 1-bit adder with two half adders is technology-specific, because it is unlikely that such a method would exist unless there was a half adder cell in the target cell library. Technology-specific methods are required for the component decomposition algorithm to take advantage of available library cells. I discuss this topic further in Chapter 8.

```
<[p0_00,p0_01,p0_02],
 [p0_03,p0_04],
 [ <A,p0_00,[p1_00]>,<B,p0_01,[p1_01]>,<C0,p0_02,[p2_01]>,<S,p2_02,[p0_03]>,
   <C1,p3_02,[p0_04]>,<P,p1_02,[p2_00]>,<X,p1_03,[p3_00]>,<Y,p2_03,[p3_01]> ],
 [ ],
 [ <HA, <ADD, [<I0,1>,<I1,1>], [<O0,1>,<COUT,1>], [ ]>,
   <OR, <OR, [<I0,1>,<I1,1>], [<O0,1>], [ ]>,
 [ <HA_0, <[p1_00,p1_01], [p1_02,p1_03], <ADD, [<I0,1>,<I1,1>], [<O0,1>,<COUT,1>], [ ]>, >>,
   <HA_1, <[p2_00,p2_01], [p2_02,p2_03], <ADD, [<I0,1>,<I1,1>], [<O0,1>,<COUT,1>], [ ]>, >>,
   <OR_0, <[p3_00,p3_01], [p3_02], <OR, [<I0,1>,<I1,1>], [<O0,1>], [ ]>, >> ] >
```

(a)



(b)

Figure 5.6: Netlist for 1-bit adder: (a) textual form; and (b) graphical depiction.

```
OR2:  <<OR,[<I0,1>,<I1,1>],[<O0,1>],[ ]>, ,[<NAME,OR2>,<AREA,...>,<DELAY,...>,<COST,...>]>
```

(a)

```
HA:   <<HA,[<I0,1>,<I1,1>],[<O0,1>,<COUT,1>],[ ]>, ,[<NAME,HA>,<AREA,...>,<DELAY,...>,<COST,...>]>
```

(b)

```
ADD1: <<ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[ ]>,
       < [p0_00,p0_01,p0_02],
         [p0_03,p0_04],
         [ <A,p0_00,[p1_00]>,<B,p0_01,[p1_01]>,<C0,p0_02,[p2_01]>,<S,p2_02,[p0_03]>,
           <C1,p3_02,[p0_04]>,<P,p1_02,[p2_00]>,<X,p1_03,[p3_00]>,<Y,p2_03,[p3_01]> ],
         [ <HA, <ADD, [<I0,1>,<I1,1>], [<O0,1>,<COUT,1>], [ ]>>,
           <OR, <OR, [<I0,1>,<I1,1>], [<O0,1>], [ ]>>,
         [ <HA_0, <[p1_00,p1_01], [p1_02,p1_03], <ADD, [<I0,1>,<I1,1>], [<O0,1>,<COUT,1>], [ ]>,   HA>>,
           <HA_1, <[p2_00,p2_01], [p2_02,p2_03], <ADD, [<I0,1>,<I1,1>], [<O0,1>,<COUT,1>], [ ]>,   HA>>,
           <OR_0, <[p3_00,p3_01], [p3_02], <OR, [<I0,1>,<I1,1>], [<O0,1>], [ ]>,   OR2>>]>,
       [ ]>
```

(c)

```
ADD4: <<ADD,[<I0,4>,<I1,4>,<CIN,1>],[<O0,4>,<COUT,1>],[ ]>,
       < [p0_00,p0_01,p0_02,p0_03,p0_04,p0_05,p0_06,p0_07,p0_08],
         [p0_09,p0_10,p0_11,p0_12,p0_13],
         [ <A0,p0_00,[p1_00]>,<A1,p0_01,[p2_00]>,<A2,p0_02,[p3_00]>,<A3,p0_03,[p4_00]>,
           <B0,p0_04,[p1_01]>,<B1,p0_05,[p2_01]>,<B2,p0_06,[p3_01]>,<B3,p0_07,[p4_01]>,
           <S0,p1_03,[p0_09]>,<S1,p2_03,[p0_10]>,<S2,p3_03,[p0_11]>,<S3,p4_03,[p0_12]>,
           <C0,p0_08,[p1_02]>,<C1,p1_04,[p2_02]>,<C2,p2_04,[p3_02]>,<C3,p3_04,[p4_02]>,
           <C4,p4_04,[p0_13]> ],
         [ <ADD_1,<ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[ ]>> ],
         [ <ADD_1_0, <[p1_00,p1_01,p1_02], [p1_03,p1_04], <ADD, [<I0,1>,<I1,1>,<CIN,1>], [<O0,1>,<COUT,1>], [ ]>,   ADD1 >>,
           <ADD_1_1, <[p2_00,p2_01,p2_02], [p2_03,p2_04], <ADD, [<I0,1>,<I1,1>,<CIN,1>], [<O0,1>,<COUT,1>], [ ]>,   ADD1 >>,
           <ADD_1_2, <[p3_00,p3_01,p3_02], [p3_03,p3_04], <ADD, [<I0,1>,<I1,1>,<CIN,1>], [<O0,1>,<COUT,1>], [ ]>,   ADD1 >>,
           <ADD_1_3, <[p4_00,p4_01,p4_02], [p4_03,p4_04], <ADD, [<I0,1>,<I1,1>,<CIN,1>], [<O0,1>,<COUT,1>], [ ]>,   ADD1 >>]>,
       [ ]>
```

(d)

Figure 5.7: Fully-mapped netlist for a 4-bit adder: (a) OR2; (b) HA; (c) ADD1; and (d) ADD4.

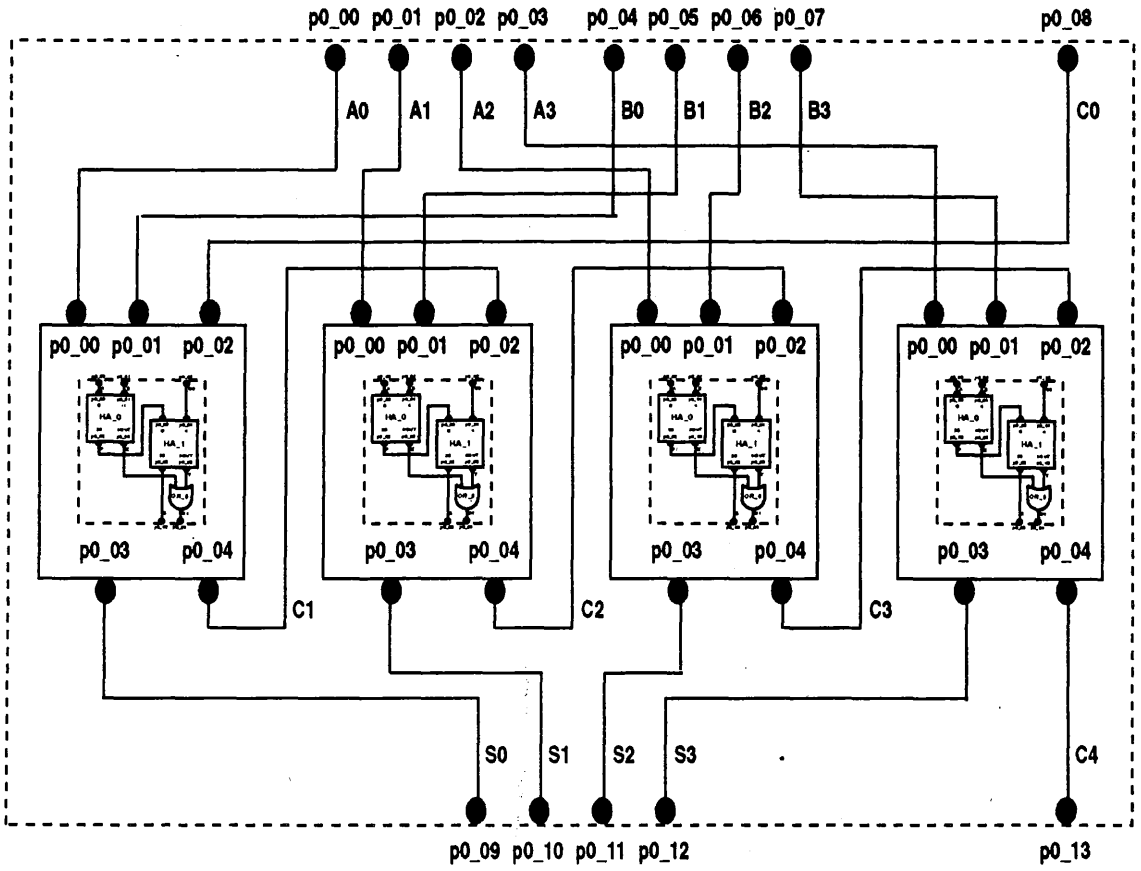


Figure 5.8: Graphical depiction of fully-mapped netlist for a 4-bit adder

5.3 Exploring Design Alternatives

One aspect of the component decomposition algorithm that is not illustrated by this example is that of design space expansion by generating alternative implementations of the same specification. Without entering the same level of detail as above, I can demonstrate how this is done by extending the library of decomposition methods with the three methods shown in Figures 5.9(a), 5.10(a), and 5.11(a), which are depicted graphically in Figures 5.9(b), 5.10(b), and 5.11(b), respectively. These methods can be used to generate alternative implementations of an ?n-bit adder in which the level of look ahead varies from 0, which is a full ripple-carry adder, to $\log(n, 4)$, which is a full carry look-ahead (CLA) adder.

The method in Figure 5.9(a) is applicable to any n-bit adder with no attributes (e.g., no specification of style or levels of look ahead). This method implements the adder with an n-bit ripple-carry adder defined by the component specification

```
<ADD,[<IO,?n>,<I1,?n>,<CIN,1>],[<OO,?n>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,?l>]>
```

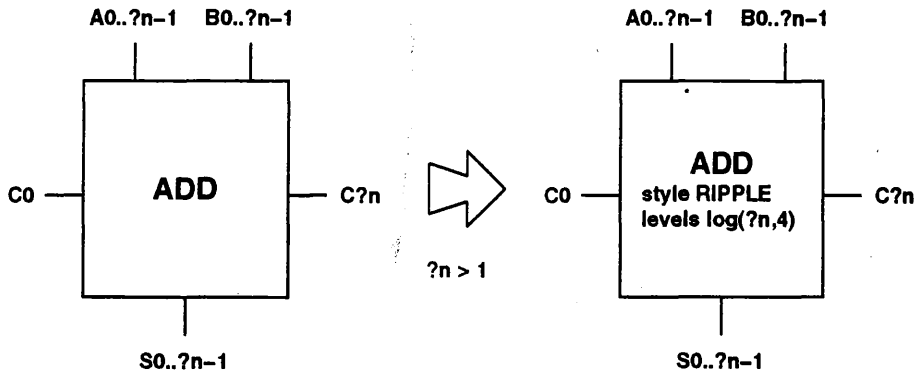
In this specification, the attribute `<LEVELS,?l>` indicates that the adder is to be implemented using *at most* `?l` levels of carry look-ahead, where `?l` is the levels of look ahead possible in a full `?n`-bit CLA adder, assuming a 4-bit CLA generator.

The two methods seen in Figures 5.10(a) and 5.11(a) define alternative implementations of an `?n`-bit ripple-carry adder with at most `?l` levels of look-ahead. The method in Figure 5.10(a) implements the adder with exactly `?l` levels of look-ahead. In this method, `?m` is bound to the width of a full CLA adder with exactly `?l` levels of look-ahead, and `?n/?m` of these adders are cascaded to implement the `?n`-bit ripple adder. The method in Figure 5.11(a) implements the adder recursively with at most `?l-1` levels of look-ahead.

Given these methods, the component decomposition algorithm can produce a series of alternative designs for a specified adder. For instance, if the input netlist requires a 64-bit adder, the component decomposition algorithm will generate four alternative designs: a ripple adder with 3 levels of look-ahead, which is a full CLA adder; with 2 levels of look-ahead, which ripples four 16-bit CLA adders; with 1 level of look-ahead, which ripples sixteen 4-bit CLA adders; and with 0 levels of look-ahead, which is a full ripple-carry adder.

```
<<ADD,[<I0,?n>,<I1,?n>,<CIN,1>],[<O0,?n>,<COUT,1>],[ ]>,
 [ ?n > 1 ],
 [
  LOOP( [ STEP(?i,0,?n-1,1) ],
    [
     CONNECT-SRC(CONCAT(A,?i), NETLIST, I0, ?i),
     CONNECT-SRC(CONCAT(B,?i), NETLIST, I1, ?i),
     CONNECT-SNK(CONCAT(S,?i), NETLIST, O0, ?i)
    ]),
  CONNECT-SRC(C0, NETLIST, CIN, 0),
  CONNECT-SNK(CONCAT(C,?n), NETLIST, COUT, 0),
  BIND(?l, CEILING(LOG(?n, 4))),
  BIND(?module, CONCAT(ADD-, ?n, _RIPPLE_, ?l)),
  ADD-CSPEC(ADDN, <ADD,[<I0,?n>,<I1,?n>,<CIN,1>],[<O0,?n>,<COUT,1>],
                                [<STYLE,RIPPLE>,<LEVELS,?l>]>),
  ADD-MODULE(?module, ADDN),
  LOOP( [ STEP(?i,0,?n-1,1) ],
    [
     CONNECT-SNK(CONCAT(A,?i), ?module, I0, ?i),
     CONNECT-SNK(CONCAT(B,?i), ?module, I1, ?i),
     CONNECT-SRC(CONCAT(S,?i), ?module, O0, ?i)
    ]),
  CONNECT-SNK(C0, ?module, CIN, 0),
  CONNECT-SRC(CONCAT(C,?n), ?module, COUT, 0)
 ]>
```

(a)

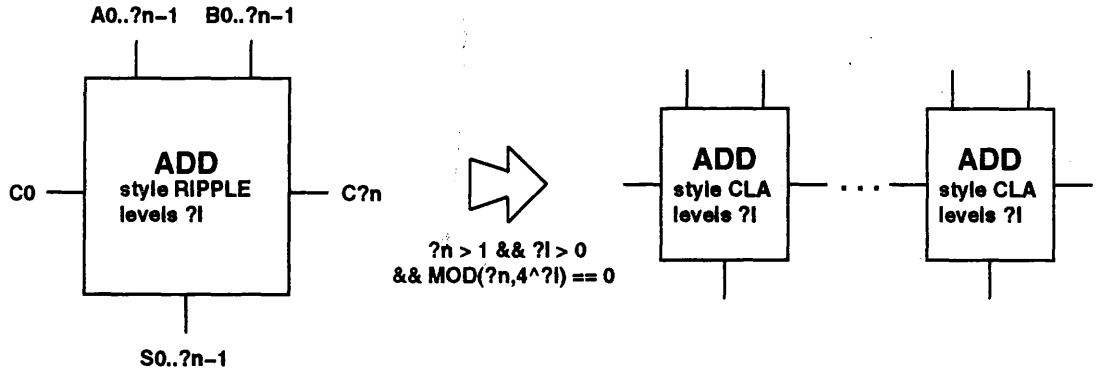


(b)

Figure 5.9: Top-level method for adder decomposition: (a) textual specification; and (b) graphical depiction.

```
<<ADD,[<I0,?n>,<I1,?n>,<CIN,1>],[<O0,?n>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,?l>]>,
 [ ?n > 1 && ?l > 0 && MOD(?n,4^?l) == 0 ],
 [
  LOOP( [ STEP(?i,0,?n-1,1) ],
    [
     CONNECT-SRC(CONCAT(A,?i), NETLIST, I0, ?i),
     CONNECT-SRC(CONCAT(B,?i), NETLIST, I1, ?i),
     CONNECT-SNK(CONCAT(S,?i), NETLIST, O0, ?i)
    ]),
  CONNECT-SRC(C0, NETLIST, CIN, 0),
  CONNECT-SNK(CONCAT(C,?n), NETLIST, COUT, 0),
  BIND(?m,?l-1),
  BIND(?cspec, CONCAT(ADD-, ?m)),
  ADD-CSPEC(?cspec, <ADD,[<I0,?m>,<I1,?m>,<CIN,1>],[<O0,?m>,<COUT,1>],
                                  [<STYLE,CLA>,<LEVELS,>l>]>),
  LOOP( [ STEP(?i,0,?n-?m,?m) ],
    [
     BIND(?module, CONCAT(?cspec, _, ?k)),
     ADD-MODULE(?module, ADD-1),
     LOOP([STEP(?j,?i,?i+?m-1,1), STEP(?k,0,?m-1,1)],
       [
        CONNECT-SNK(CONCAT(A,?j), ?module, I0, ?k),
        CONNECT-SNK(CONCAT(B,?j), ?module, I1, ?k),
        CONNECT-SRC(CONCAT(S,?j), ?module, O0, ?k),
       ]),
     CONNECT-SNK(CONCAT(C,?i), ?module, CIN, 0),
     CONNECT-SRC(CONCAT(C,?i+?m), ?module, COUT, 0)
    ]),
 ] >
```

(a)

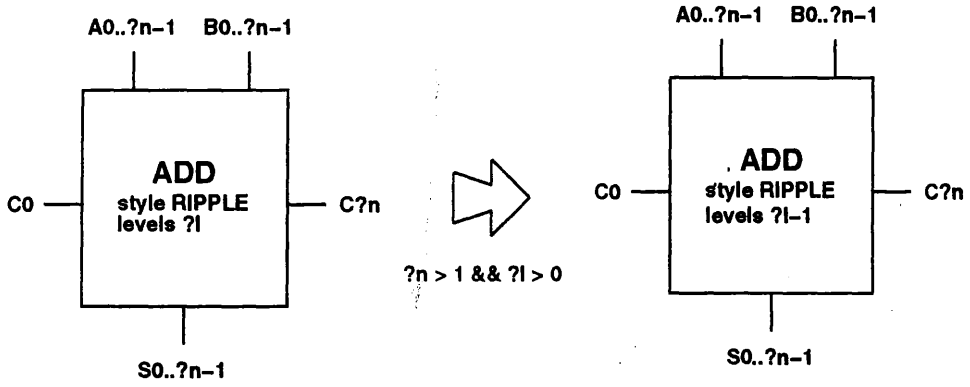


(b)

Figure 5.10: Method for adder with exactly ?l levels of look-ahead: (a) textual specification; and (b) graphical depiction.

```
<<ADD,[<I0,?n>,<I1,?n>,<CIN,1>],[<O0,?n>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,?l>]>,
 [ ?n > 1 && ?l > 0 ],
 [
  LOOP( [ STEP(?i,0,?n-1,1) ],
     [
       CONNECT-SRC(CONCAT(A,?i), NETLIST, I0, ?i),
       CONNECT-SRC(CONCAT(B,?i), NETLIST, I1, ?i),
       CONNECT-SNK(CONCAT(S,?i), NETLIST, O0, ?i)
     ]),
  CONNECT-SRC(C0, NETLIST, CIN, 0),
  CONNECT-SNK(CONCAT(C,?n), NETLIST, COUT, 0),
  ADD-CSPEC(ADDN, <ADD,[<I0,?n>,<I1,?n>,<CIN,1>],[<O0,?n>,<COUT,1>],
                                  [<STYLE,RIPPLE>,<LEVELS,?l-1>]>),
  BIND(?module, CONCAT(ADD-, ?n, _RIPPLE_, ?l-1)),
  ADD-MODULE(?module, ADDN),
  LOOP( [ STEP(?i,0,?n-1,1) ],
     [
       CONNECT-SNK(CONCAT(A,?i), ?module, I0, ?i),
       CONNECT-SNK(CONCAT(B,?i), ?module, I1, ?i),
       CONNECT-SRC(CONCAT(S,?i), ?module, O0, ?i)
     ]),
  CONNECT-SNK(C0, ?module, CIN, 0),
  CONNECT-SRC(CONCAT(C,?n), ?module, COUT, 0)
 ]>
```

(a)



(b)

Figure 5.11: Method for adder with less than ?l levels of look-ahead: (a) textual specification; and (b) graphical depiction.

Chapter 6
The DTAS Component Generation System

In this chapter, I present an implementation of the component decomposition algorithm called the Design and Technology Adaptation System (DTAS). First, I overview the system architecture; then, I describe the DTAS design language, relating it to the component decomposition algorithm and highlighting "ease of use" features; next, I outline the support features found in the DTAS design environment; finally, I discuss aspects of technology independence as they relate to DTAS. A reference manual for DTAS appears in Kipps (1991).

6.1 Overview

DTAS is a component generation system for RT datapath components. This includes *combinational components*, such as decoders, multiplexers, parity checkers, and function generators, *arithmetic components*, such as adders, comparators, multipliers, and ALUs, and *sequential components*, such as shift registers and counters. Designs are hierarchically decomposed into netlists of layout cells from a given ASIC library. These can be simple Boolean cells or complex functional cells, from multiplexers, adders, and comparators up to n-bit ALUs, multipliers, and counters. DTAS compares alternative design styles to find candidate designs that best fit performance constraints. Designs can be output in structural VHDL and input to logic synthesis and layout tools.

The implementation of DTAS is based on the component decomposition algorithm, as defined in Chapter 4. DTAS has additional "easy-of-use" features not defined in the algorithm, include component "type" declarations for consistency checking, component decomposition described in terms of Boolean expressions and function tables, iterative expansion of decomposition methods, and port types and attributes plus special recognition of control ports for multiple-operation components, such as ALUs.

DTAS is implemented as a *design language*, which follows naturally from the definition of the component decomposition algorithm. Decomposition methods constitute the subroutines that the language executes. DTAS also contains a support environment for measuring the area and delay, for collecting empirical results, and for interfacing to other synthesis tools.

6.2 System Architecture

The top-level structure of DTAS is outlined in Figure 6.1. The input to DTAS is a technology-independent structural netlist of RTL components, described using the GENUS component library (Dutt, 1988). DTAS also accepts as input individual GENUS components specifications.

The input netlist is passed through a phase of functional decomposition and technology mapping. The result is a set of hierarchical, library-specific netlists that represent alternative implementations of the components in the input netlist. Each output netlist traces the top-down design of the input netlist into subcomponents. Leaves implement the design with cells drawn from the given ASIC library. Netlists can be output in structural VHDL (hierarchical or flat) and input to logic synthesis or layout tools.

DTAS is implemented as a constructive language, the architecture of which is shown in Figure 6.2. This includes a parser for reading and loading decomposition methods from text files and an interpreter for selecting and firing methods. Each method is described using an abstracted syntax over that defined in Chapter 4. Decomposition can also be described with a combination of Boolean description and connected subcomponents.

6.3 The DTAS Design Language

The objective behind the design of the DTAS design language was to develop an concise syntax that was sufficiently expressive for describing the data structures of the component decomposition algorithm without a loss of precision. In this section, I overview the DTAS design language and relate it to the data structures and functions of the component decomposition algorithm defined in Chapter 4. The syntax and semantics of the DTAS design language are described in detail in Kipps (1991).

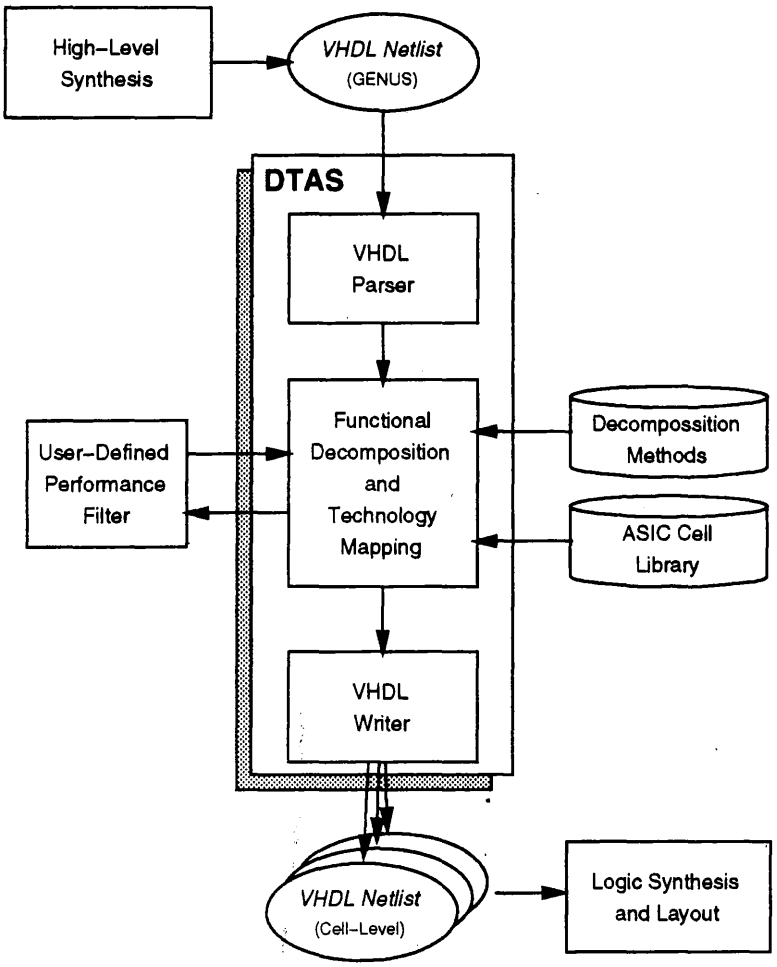


Figure 6.1: Top-level structure of DTAS.

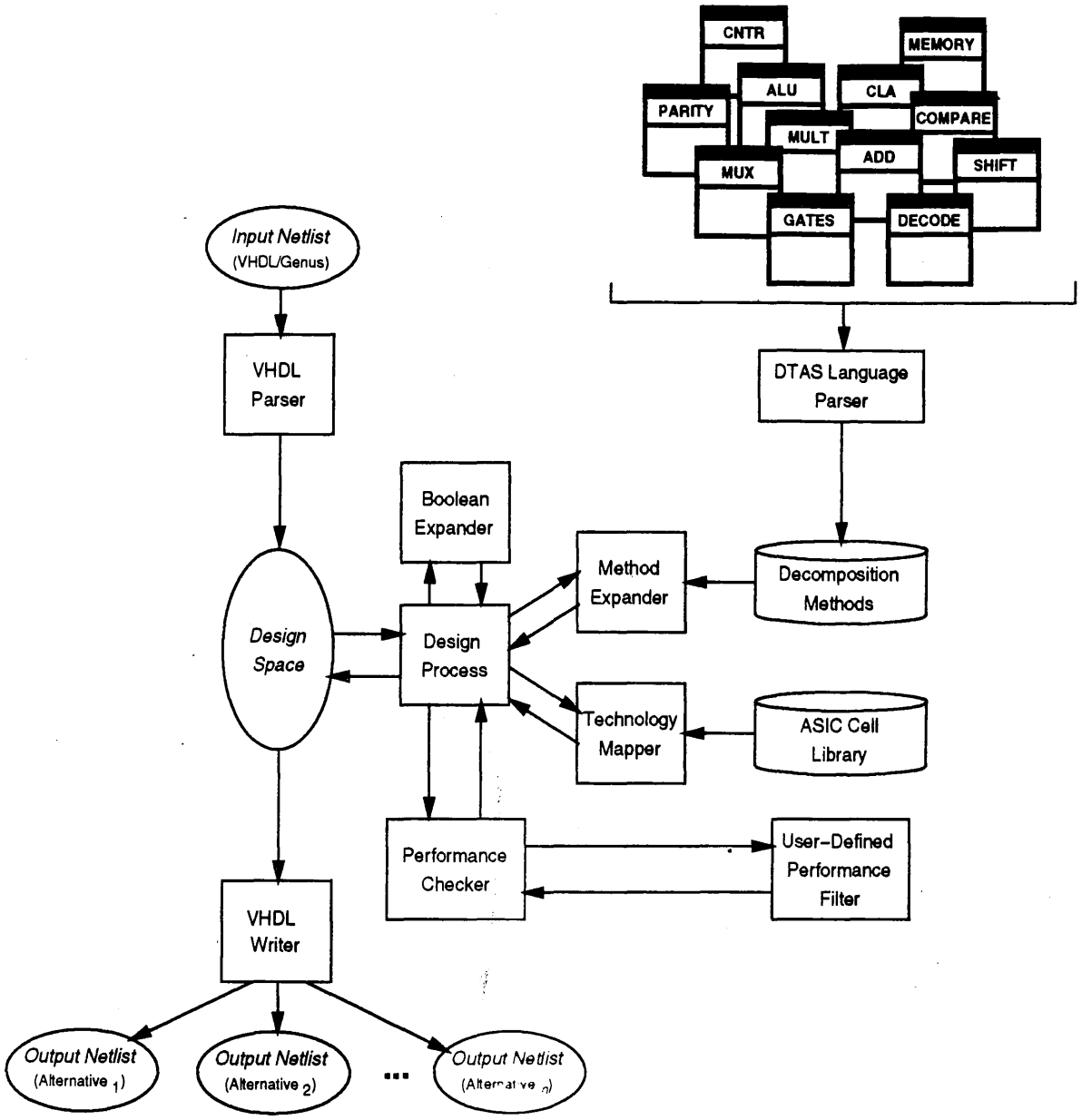


Figure 6.2: System architecture of DTAS design language.

6.3.1 Fundamental Syntactic Form

The fundamental syntactic form of the DTAS design language is shown in Figure 6.3(a). This form is used for a number of component specification tasks. *Type* is the function type of the component. The two parenthesized sequences of *pspec*s (port specifiers) identify the input and output ports of the specification, respectively. The *pdecl*s (port declarations) assign port types (*ptype*), such as DATA, CLOCK, CARRY, and CONTROL, and attributes to named ports. The *attr*s define the attribute/value pairs of the specification. Variations on this syntactic form are used to specify modules, to define the heads of decomposition methods, as well as to specify modules within the body of methods, and to declare component types, which define the space of legal component specifications.

6.3.2 Module Specification

When the syntactic form shown in Figure 6.3(a) is used to specify a module, port specifiers serves two functions. First, they identify ports of the module's component specification. Second, they define a mapping between the pins of the module corresponding to those ports and the wires to be connected to those pins. In particular, the port specifier of a module has the general form:

[*pname* =>] *wspec* [^ *width*]

where square brackets denote optionality. *Pname* is the name of the port, and *width* is its width; *wspec* (wire specifier) identifies the wires to be connected to the corresponding pins of the port. *Width* defaults to one. When not supplied, *pname* is inferred from a matching component type, as discussed later in this section. A wire specifier (*wspec*) normally has one of four forms:

name | *name*[*range*] | *name*{.*range*}$^+$ | <>

where bars separate alternatives. The first form identifies all wires of a *name*'ed range; the second two forms identify a subsequence of these wires defined by *range*; the fourth form designates a empty wire sequence, which is only used for optional ports. The sequence of wires identified by the *wspec* will be connected to the corresponding pins of the identified port.

Using this syntax, a netlist containing a single 4-bit adder module can be described as shown in Figure 6.3(b). This can be compared to the netlist specification of the component decomposition algorithm, seen earlier in Figure 5.1 and shown again in Figure 6.3(c). In Figure 6.3(b), `A^4`, `B^4`, `C.0`, `S^4`, and `C.4` are port specifiers. The specifier `A^4` identifies an unnamed 4-bit port whose corresponding pins will be connected to wires `A0`, `A1`, `A2`, and `A3`, as shown in Figure 6.3(c). Likewise, the port specifier `C.0` identifies an unnamed 1-bit port; its corresponding pin will be connected to wire `C0`.

```
type ( pspec* )
     ( pspec* )
  pdelcs*
  attr*

pdecl ::= :ptype ( pname( attrs* ) )+

attr  ::= :name value
```

(a)

```
ADD (A^4 B^4 C.0)
    (S^4 C.4)
  :style RIPPLE
```

(b)

```
< [p0_00,p0_01,p0_02,p0_03,p0_04,p0_05,p0_06,p0_07,p0_08],
  [p0_09,p0_10,p0_11,p0_12,p0_13],
  [ <A0,p0_00,[p1_00]>,<A1,p0_01,[p1_01]>,<A2,p0_02,[p1_02]>,<A3,p0_03,[p1_03]>,
    <B0,p0_04,[p1_04]>,<B1,p0_05,[p1_05]>,<B2,p0_06,[p1_06]>,<B3,p0_07,[p1_07]>,
    <C0,p0_08,[p1_08]>,<S0,p1_09,[p0_09]>,<S1,p1_10,[p0_10]>,<S2,p1_11,[p0_11]>,
    <S3,p1_12,[p0_12]>,<C4,p1_13,[p0_13]> ],
  [ ],
  [ <ADD_4,<ADD,[<I0,4>,<I1,4>,<CIN,1>],[<O0,4>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,0>]>> ],
  [ <ADD_4_0,  <[p1_00,p1_01,p1_02,p1_03,p1_04,p1_05,p1_06,p1_07,p1_08],
                [p1_09,p1_10,p1_11,p1_12,p1_13],
                <ADD,[<I0,4><I1,4>,<CIN,1>],[<O0,4>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,0>]>, >> ] >
```

(c)

```
ADD (I0^?n I1^?n [CIN])
    (O0^?n [COUT])
  :data I0 I1 O0
  :carry CIN COUT
  :style {RIPPLE|CLA}
  :levels ?l  (default: 0)
  where integerp(?l)
```

(d)

Figure 6.3: DTAS design language examples: (a) fundamental syntactic form; (b) sample DTAS netlist specification; (c) formal representation; and (d) sample component type.

6.3.3 Component Types

Component types define the legal set of component specifications. If a component specification does not match any declared component type, DTAS calls an error. When a component specification does match a component type, then portions of component specification that were undefined are inferred from the component type. For instance, the names of the unnamed ports in Figure 6.3(b) are inferred by matching the component specification of the 4-bit adder module against the component type shown in Figure 6.3(d). Also inferred is the attribute `<LEVELS,0>`.

A component type is essentially a component specification pattern, much like the head of a decomposition method. Port widths and attributes can have variables used in the place of literal values. In addition, attributes can be defined as optional or as having default values. A component type can also be constrained by a test on the values bound to its variables. As seen in Figure 6.3(d), the test of a component type is introduced by the symbol **where**.

When the syntactic form shown in Figure 6.3(a) is used to declare a component type, port specifiers define the names of ports and define which ports are optional. In particular, the port specifier of a component type has one of two forms:

pname [^ *width*] | [*pname* [^ *width*]].

Pname is the name of the port, and *width* is its width. The first form defines the port as mandatory; the second as optional. Optional port specifiers match empty ports of the form `<>` when they appear in a component specification. For instance, the port list `(A^4 B^4 <>)` matches `(I0^?n I1^?n [CIN])`; trailing optional port specifiers do not require corresponding empty ports, e.g., `(A^4 B^4)` also matches `(I0^?n I1^?n [CIN])`.

Finally, the syntax of variables in the DTAS design language allows implicit conditions to be added to the test of a component type. For instance, in Figure 6.3(d), the value of attribute **style** is `{RIPPLE|CLA}`, which defines an unnamed variable, the value of which either must be the symbol `RIPPLE` or the symbol `CLA`. These constraints are added to the test of the component type.

Given all of the above, the component type shown in Figure 6.3(d) will match any component specification for a n-bit adder, with or without carry input and output. Matching component specifications must have an attribute **style**, whose value must be one of `RIPPLE` or `CLA` and, optionally, an attribute **levels**, whose value is an integer that defaults to zero.

```
ADD (A B)
    (S CO)
  :name HA1
  :load (2 3)
  :area 5
  :delay      |          S                    CO
              +-------------------------------------------
            A | 1.08+0.14:0.94+0.07    0.048+0.14:0.77+0.05
            B | 1.08+0.14:0.94+0.07    0.048+0.14:0.77+0.05

  -> let S   := A(+)B
        CO  := A*B
```

(a)

```
GATE OR2      1     Z=A+B
   PIN * NONINV   1  999  0.38  0.1443  1.35  0.0788
```

(b)

Figure 6.4: DTAS design language examples: library cells (a) half adder; and (b) OR-gate.

6.3.4 Library Cells

Library cells are also specified using a variation on the syntactic form from Figure 6.3(a). Cells with a single output, such as Boolean gates, can also be specified using the misII cell format (Lisanke, 1988). Examples of both syntactic forms are shown in Figures 6.4(a) and (b), respectively.

When the syntactic form from Figure 6.3(a) is used to specify a library cell, as in Figure 6.4(a), port specifiers have much the same meaning as when used in specifying modules; namely, they identify ports and relate corresponding pins to wires. The syntax of port specifiers is also the same, i.e.,

$$[\ pname\ \texttt{=>}\ ]\ wspec\ [\ \texttt{\^{}}\ width\ ].$$

As shown in Figure 6.4(a), the wires of *wspec* are used to specify delay tables, which list pin-to-pin delays through the cell, and to describe the cells behavior with Boolean equations or function tables. Attributes `name`, `load`, `area`, `delay`, and the arrow (`->`) are recognized by DTAS as properties of the cell.

6.3.5 Decomposition Methods

In the DTAS design language, a decomposition method has the general syntactic form shown in Figure 6.5(a). The head, test, and iteration clause of a method are separated from its body by an arrow (`->`). The iteration clause, which is introduced by the symbol **varying**, is another adaptation on the component decomposition algorithm and is explained later in this section.

In the body of the method, actions are grouped with curly braces. In addition, BIND actions have the general form:

let { *var* := *expression* }$^{+}$,

CASE actions have the general form:

if *cond action* { **else** *action* }

and LOOP actions have the general form:

for *iter* { **as** *iter* }* *action*

The ADD-CSPEC, ADD-MODULE, CONNECT-SRC, and CONNECT-SNK actions are implicit in the syntax of the decomposition method.

The head and body of a decomposition method uses the syntactic form seen earlier in Figure 6.3(a). As when used to specify a method, port specifiers both identify ports and define a mapping between pins and wires. Port specifiers also have the same general syntactic form:

[*pname* =>] *wspec* [^ *width*]

with two exceptions. First, when used in the head of a method, the ports correspond to the ports of matched component specifications; the corresponding pins belong to the netlist under construction. Port specifiers can delimited with square brackets to denote optionality. Second, both in the head and body, *width* can be a variable or expression containing variables (as well as a literal), even in the head of the method.

A sample DTAS decomposition method is shown in Figure 6.5(b). This method subsumes the decomposition method of the component decomposition algorithm seen earlier in Figure 5.4 and shown again in Figure 6.5(c). Not only is this method applicable to the component specification of the module seen earlier in Figure 5.1(b), it is also applicable to n-bit adders without a carry input or carry output.

One more example of DTAS syntax for decomposition methods is shown in Figure 6.6. The method appearing in Figure 6.6(a) decomposes a 1-bit adder into two half adders. The carry output is defined with a Boolean equation. This method is equivalent to the method of the component decomposition algorithm seen earlier in Figure 5.9 and shown again in Figure 6.6(b).

```
head
  [ where test ]
  [ varying iter { as iter }* ]
  ->
     body
```

(a)

```
ADD (A^?n B^?n [C.0])
    (S^?n [C.?n])
  :style RIPPLE
  :levels 0
  where ?n > 1
  ->
     for ?i from 0 to ?n-1
       ADD (A[?i] B[?i] C.?i)
           (S[?i] C.?i+1)
```

(b)

```
<<ADD,[<I0,?n>,<I1,?n>,<CIN,1>],[<O0,?n>,<COUT,1>],[<STYLE,RIPPLE>,<LEVELS,0>]>,
 [ ?n > 1 ],
 [
  LOOP( [ STEP(?i,0,?n-1,1) ],
     [
       CONNECT-SRC(CONCAT(A,?i), NETLIST, I0, ?i),
       CONNECT-SRC(CONCAT(B,?i), NETLIST, I1, ?i),
       CONNECT-SNK(CONCAT(S,?i), NETLIST, O0, ?i)
     ]),
  CONNECT-SRC(C0, NETLIST, CIN, 0),
  CONNECT-SNK(CONCAT(C,?n), NETLIST, COUT, 0),

  ADD-CSPEC(ADD-1, <ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[ ]>),

  LOOP( [ STEP(?i,0,?n-1,1) ],
     [
       ADD-MODULE(CONCAT(ADD-1_, ?i), ADD-1),
       CONNECT-SNK(CONCAT(A,?i), CONCAT(ADD-1_, ?i), I0, 0),
       CONNECT-SNK(CONCAT(B,?i), CONCAT(ADD-1_, ?i), I1, 0),
       CONNECT-SNK(CONCAT(C,?i), CONCAT(ADD-1_, ?i), CIN, 0),
       CONNECT-SRC(CONCAT(S,?i), CONCAT(ADD-1_, ?i), O0, 0),
       CONNECT-SRC(CONCAT(C,?i+1), CONCAT(ADD-1_, ?i), COUT, 0)
     ]),
 ]>
```

(c)

Figure 6.5: DTAS design language examples: (a) decomposition method syntax; (b) sample decomposition method; and (c) formal representation.

An alternative decomposition method for a 1-bit adder is shown in Figure 6.6(c). This method demonstrates the use of Boolean equations in describing the body of a decomposition method. This method is applicable to full and half adders, as well as 1-bit adders with a carry propagate (P) and generate (G), i.e., for use with a carry look-ahead generator. When this method is expanded, the Boolean equations are minimized using tabulation and mapped directly in to generic Boolean components. The advantage of using a Boolean description for components such as a 1-bit adder is flexibility. When optional output ports, such as carry output and carry propagate and generate, are not specified, the corresponding equations are discarded during minimization (unless used as intermediate equations). Likewise, when optional input ports, such as carry input, are not specified, their default literal value (zero or user defined) is used in their place during minimization. Thus, when this method is applied to the specification of a half adder, it only expands into the logic needed for the half adder. This means that a single decomposition method can be used for a range of base (Boolean) implementations without generating spurious circuits.

The next example, shown in Figure 6.7, demonstrates the DTAS syntax for defining methods that search the space of design alternatives. This is done with *iterative method expansion.* The method shown in Figure 6.7(a) is applicable to a component specification for an n-bit adder, with or without carry enable and with no attributes. The statement sandwiched between the test and the arrow,

```
varying ?l from ceiling(log(?n,4)) downto 0,
```

causes this method to be expanded multiple times, i.e., once for each step of `?l`. This has the same effect as the two decomposition methods seen earlier in Figures 5.9 and 5.11, the first of which is shown again in Figure 6.7(b).

```
ADD (A B C.0)
    (S C.1)
  ->
    ADD (A B)
        (P X)
    ADD (P C.0)
        (S Y)
    let C.1 := X+Y
```

(a)

```
ADD(A B [C.0])
    (S [C.1] [P] [G])
  ->
    let P   := A(+)B
        G   := A*B
        S   := P(+)C.0
        C.1 := C.0*P + G
```

(c)

```
<<ADD,[<I0,1>,<I1,1>,<CIN,1>],[<O0,1>,<COUT,1>],[ ]>,
 [ ],
 [
  CONNECT-SRC(A, NETLIST, I0, 0),
  CONNECT-SRC(B, NETLIST, I1, 0),
  CONNECT-SRC(C0, NETLIST, CIN, 0),
  CONNECT-SNK(S, NETLIST, O0, 0),
  CONNECT-SNK(C1, NETLIST, COUT, 0),

  ADD-CSPEC(HA, <ADD,[<I0,1>,<I1,1>],[<O0,1>,<COUT,1>],[ ]>),
  ADD-CSPEC(OR, <OR,[<I0,1>,<I1,1>],[<O0,1>],[ ]>),

  ADD-MODULE(HA_0, HA),
  ADD-MODULE(HA_1, HA),
  ADD-MODULE(OR_0, OR),

  CONNECT-SNK(A, HA_0, I0, 0),
  CONNECT-SNK(B, HA_0, I1, 0),
  CONNECT-SRC(P, HA_0, O0, 0),
  CONNECT-SRC(X, HA_0, COUT, 0),

  CONNECT-SNK(P, HA_1, I0, 0),
  CONNECT-SNK(C0, HA_1, I1, 0),
  CONNECT-SRC(S, HA_1, O0, 0),
  CONNECT-SRC(Y, HA_1, COUT, 0),

  CONNECT-SNK(X, OR_0, I0, 0),
  CONNECT-SNK(Y, OR_0, I1, 0),
  CONNECT-SRC,(C1, OR_0, O0, 0)
 ]>
```

(b)

Figure 6.6: DTAS design language examples: (a) 1-bit adder to half adders; (b) formal representation; and (c) 1-bit adder using Boolean descriptions.

```
ADD (A^?n B^?n [C.0])
     (S^?n [C.?n])
  where ?n > 1
  varying ?l from ceiling(log(?n,4)) downto 0
  ->
     ADD (A^?n B^?n C.0)
          (S^?n C.?n)
       :style RIPPLE
       :levels ?l
```

(a)

```
< <ADD,[<I0,?n>,<I1,?n>,<CIN,1>],[<O0,?n>,<COUT,1>],[ ]>,
  [ ?n > 1 ],
  [
    LOOP( [ STEP(?i,0,?n-1,1) ],
      [
        CONNECT-SRC(CONCAT(A,?i), NETLIST, I0, ?i),
        CONNECT-SRC(CONCAT(B,?i), NETLIST, I1, ?i),
        CONNECT-SNK(CONCAT(S,?i), NETLIST, O0, ?i)
      ]),

    CONNECT-SRC(C0, NETLIST, CIN, 0),
    CONNECT-SNK(CONCAT(C,?n), NETLIST, COUT, 0),

    BIND(?l, CEILING(LOG(?n, 4))),
    BIND(?module, CONCAT(ADD-, ?n, _RIPPLE_, ?l)),

    ADD-CSPEC(ADDN, <ADD,[<I0,?n>,<I1,?n>,<CIN,1>],[<O0,?n>,<COUT,1>],
                        [<STYLE,RIPPLE>,<LEVELS,?l>]>),
    ADD-MODULE(?module, ADDN),

    LOOP( [ STEP(?i,0,?n-1,1) ],
      [
        CONNECT-SNK(CONCAT(A,?i), ?module, I0, ?i),
        CONNECT-SNK(CONCAT(B,?i), ?module, I1, ?i),
        CONNECT-SRC(CONCAT(S,?i), ?module, O0, ?i)
      ]),

    CONNECT-SNK(C0, ?module, CIN, 0),
    CONNECT-SRC(CONCAT(C,?n), ?module, COUT, 0)
  ]>
```

(b)

Figure 6.7: DTAS design language examples: (a) n-bit adder; and (b) formal representation.

6.3.6 Control Ports on Multiple-Operation Components

The final example of the DTAS design language demonstrates the syntax of control ports on multiple-operation components, such as multiplexers and ALUs, that require an "operation select" input. For brevity, this example focuses on multiplexers.

A multiplexer (MUX) is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a number of select lines. Typically, there are n select lines for 2^n input lines, but there can also be a one to one correspondence between select lines and input lines. MUXs can be specified in the DTAS design language using the constructs shown in Figure 6.8.

The two component types shown in Figure 6.8(a) and (b) define the range of legal MUX component specifications. Both component types represent the input lines of a MUX with port `I`. The syntax

```
I^?m:?n
```

says that port `I` has a width of `?m`×`?n` and corresponds to `?m` groups for pins with `?n` pins in each group; in other words, `I` represents `?m` `?n`-bit input lines. The single output line is represented by port `O` of width `?n`. The syntax

```
(0..?s-1)
```

designates a sequential list of integers from a lower bound (0) to an upper bound (`?s-1`), inclusive. The syntax

```
?idx|#'listp
```

is a constraint that adds the condition (`listp ?idx`) to the test of each component type.

The select lines are represented by the input port `SEL` of width `?s`, where `SEL` is defined to be a control (`ctrl`) port. In DTAS, the select logic of a component is defined in terms of the input signals that can be applied to the pins corresponding to a control port. A control port is specially recognized by DTAS as having three attributes: **keys**, **style**, and **mappings**. The value of `keys` is a list of values representing the things that can be selected, ordered from low signal combination to high. The value of **style** is one of **UNARY** or **BINARY**, which designates how signal combinations are encoded to make the select logic. The value of `mappings` is a list of pairs associating each key with the select logic for recognizing a distinct signal combination. The value of **keys** must be defined by the user; the value of **style** may be defined but can also be inferred from the number of keys and the width of the control port; the value of **mappings** is always assigned by DTAS.

```
MUX (I^?m:?n SEL^?m)
     (O^?n)
   :data I O
   :ctrl SEL (:keys (0..?m-1) :style UNARY)
   :select ?idx|#'listp  (default: (0..?m-1))
   where ?m == length(?idx)
      && subsetp(?idx, (0..?m-1))
```

(a)

```
MUX (I^?m:?n SEL^?s)
     (O^?n)
   :data I O
   :ctrl SEL (:keys (0..2^?s-1) :style BINARY)
   :select ?idx|#'listp  (default: (0..?m-1))
   where ?m == length(?idx)
      && subsetp(?idx, (0..?m-1))
      && ?m <= 2^?s
```

(b)

```
MUX(I^?m:?n SEL^?s)
     (O^?n)
   :select ?idx
   where ?n > 1
   ->
      for ?i from 0 to ?n-1
        MUX (I[0..?m-1:?i] SEL^?s)
            (O[?i])
          :select ?idx
```

(c)

```
MUX (I^?m SEL^?s)
     (O)
   :select ?idx
   ->
      for ?i in ?idx as ?j from 0
        let I.?j := I[?i] * SEL:?j
      let O := +(I.0..?m-1)
```

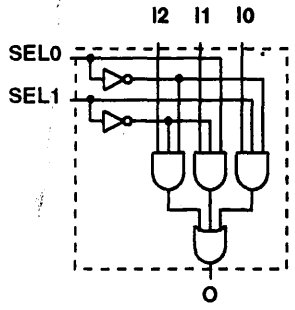
(d)

```
MUX (I^3 SEL^2)
     (O)
   :select (2 1 0)
```

(e)

SEL1	SEL0	O
0	0	I2
0	1	I1
1	0	I0

(f)



(g)

Figure 6.8: DTAS design language examples: (a) component type: MUX w/unary control; (b) component type: MUX w/binary control; (c) decomposition method: MUX w/m n-bit inputs; (d) decomposition method: m-bit MUX; (e) specification of 3-bit MUX; (f) function table for 3-bit MUX; (g) netlist implementation of 3-bit MUX.

As an example, consider a control port SEL that is used to select between four operations: addition, subtraction, increment, and decrement. The keys to this port will be the list of symbols

```
(ADD SUB INC DEC)
```

If the style of SEL is UNARY, then each distinct pin corresponding to the port controls the selection of a distinct operation; four operations means that SEL must have a width of four. Its mapping consists of the four pairs

```
<ADD, SEL[3]'*SEL[2]'*SEL[1]'*SEL[0]>
<SUB, SEL[3]'*SEL[2]'*SEL[1]*SEL[0]'>
<INC, SEL[3]'*SEL[2]*SEL[1]'*SEL[0]'>
<DEC, SEL[3]*SEL[2]'*SEL[1]'*SEL[0]'>
```

On the other hand, if the style of SEL is BINARY, then each binary combination of inputs to SEL controls the selection of a distinct operation; this means that SEL must have a width of two. Its mapping now consists of the four pairs:

```
<ADD, SEL[1]'*SEL[0]'>
<SUB, SEL[1]'*SEL[0]>
<INC, SEL[1]*SEL[0]'>
<DEC, SEL[1]*SEL[0]>
```

Other styles of encoding, such as Gray code, are possible but not implemented.

The difference between the two component types seen in Figure 6.8 is in the manner in which they define correspondence of select lines to input lines. The component type shown in Figure 6.8(a) defines SEL as a unary control port, while the component type shown in Figure 6.8(b) defines SEL as a binary control port. In both cases, the keys of SEL are list of integers from zero to the number of possible select logic encodings, i.e., ?s-1 unary encodings and $2^{?s}-1$ binary encodings. The attribute `select` designates the order in which input lines are to be selected; its value must be a list of integers from 0 to ?m-1 in any order and defaults to an order list from low to high.

The two decomposition methods shown in Figures 6.8(c) and (d) define how a MUX can be implemented. The method shown in Figure 6.8(c) is applicable to any MUX with ?m ?n-bit input lines (for ?n$>$ 1), which it implements with ?n MUXs with ?m 1-bit input lines. The syntax

```
I[0..?m-1:?i]
```

specifies a port of width ?m where the corresponding pins are connected to a subrange of wires from a group of wires named I; the source of this wire group will be the input pins of the netlist under construction; I names a group of ?m$\times$?n wires, partitioned into ?m groups of ?n wires each; the above syntax refers to a sequence of wires consisting of the ?*ith* wire from each of these ?m groups.

The method shown in Figure 6.8(d) is applicable to any MUX with `?m` 1-bit input lines. Boolean description is used to define how such MUXs are to be implemented. The Boolean term `SEL:?j` expands into the select logic associated with key `?j` of control port `SEL`.

Figure 6.8(e) shows the specification for a netlist containing a single MUX module. This module has three 1-bit input lines, represented by port `I`, and two select line, represented by `SEL`; it is not actually necessary to designate `SEL` as a binary control port, since this would be inferred from the matching component type (Figure 6.8(b)) as are its keys. After being matched against this component type, the control port `SEL` will have keys `(0 1 2 3)` and mapping

```
<0, SEL[1]'*SEL[0]'>
<1, SEL[1]'*SEL[0]>
<2, SEL[1]*SEL[0]'>
<3, SEL[1]*SEL[0]>.
```

The value of the attribute **`select`** designates that the most significant input line is to be selected by the least significant select logic encoding. This behavior is depicted by the function table shown in Figure 6.8(f).

The method in Figure 6.8(d) is applicable to the component specification from Figure 6.8(e). When expanded, the LOOP action steps `?i` through successive elements of `?idx` while incrementing `?j` starting at zero. On the first pass through the loop, `?i`=2 and `?j`=1, resulting in the expansion of the Boolean equation

```
I.0=I[2]*SEL[1]'*SEL[0]'
```

Successive passes result in the equations

```
I.1=I[1]*SEL[1]'*SEL[0]
I.2=I[0]*SEL[1]*SEL[0]'
```

The final Boolean equation ties these together at output `O`, i.e.,

```
O=I.0+I.1+I.2
```

When minimized and mapped to generic Boolean gates, these equations result in the netlist shown graphically in Figure 6.8(g).

Although not pertinent to the component decomposition algorithm, control ports are a significant feature of DTAS. Control ports decouple the things to be selected from the actual select logic. Symbolic keys are mapped automatically to their corresponding select logic independent of the logic's encoding, which means they can be reordered without complication. Within a decomposition method, the select logic associated with a control port can be accessed symbolically with a key of the port. Methods can make use of all select logic or a subset. As illustrated in Figure 6.8, it only takes two decomposition methods to implement all the possible component specifications defined by the MUX component types.

6.4 The DTAS Design Environment

DTAS is implemented in Common Lisp. This implementation supports a modest design environment for loading files of decomposition methods and cell libraries, for loading netlist data files, and for generating designs. DTAS also supports routines for computing the area and delay of fully-mapped netlists. Alternative designs can be compared by being plotted on a delay/area graph. Individual designs can be inspected graphically. Each design can be output in either of two forms of structural VHDL: one form preserves the hierarchical structure of the design; the other form flattens the design into a netlist of library cells.

6.4.1 File Types

Component type declarations, decomposition methods, library cells, and netlist data are loaded into the design environment from text files. The DTAS design environment supports a number of functions for loading different types of files in various formats.

Component types and decomposition methods that are applicable to the same component specifications are stored in SYN files. SYN files loaded into the design environment with the `rload` function. Appendix A contains listings of the SYN files used to run the experiments presented in the Chapter 7.

Library cells are stored in LIB files. The header of a LIB file names the library and designates the units of area, such as microns or equivalent 2-input NAND gates. LIB files are loaded with the `lload` function. Library cells defined using misII format are stored in MIS2LIB files and loaded with the `mload` function. Appendix B contains listings of the LIB and MIS2LIB files used in running the experiments.

Structural netlists can be specified using the syntax of the DTAS design language or in structural VHDL. The former are stored in DAT files and are loaded with the `dload` function. The latter are stored in HDL files and loaded with the `vload` function. Appendix C contains descriptions of the HDL files used in running the experiments.

6.4.2 Entering The Design Environment

The DTAS design environment is built on top of the Common Lisp and is accessed from the LISP's top-level monitor. Commands are written in LISP syntax. When you enter the DTAS design environment, you see the following

```
DESIGN AND TECHNOLOGY ADAPTATION SYSTEM (DTAS)
----------------------------------------------
 VERSION 0.7   28 AUG 1991 (c) James R. Kipps

[1] >
```

where [*i*] > is the prompt.

The files listed in Appendix A.1 have already been loaded. What needs to be loaded is the specification of cells from the target ASIC library plus library-specific methods. For instance,

```
[1] > (mload lsi)                    ; load cells in MIS2LIB format
Loading lsi.mis2lib from library...
Finished loading lsi.mis2lib

[2] > (lload lsi)                    ; load cells in LIB format
Loading LSI.LIB from library...
Finished loading LSI.LIB

[3] > (rload lsi)                    ; load library-specific methods
Loading LSI.SYN from library...
Finished loading LSI.LIB

[4] >
```

sets up the design environment to map designs into cells from the LSI macrocell library. This library, listed in Appendix B.3, includes 1-, 2-, and 4-bit adders, 4-bit carry look-ahead generators (CLAs), and other large cells. The 4-bit adders and CLAs are LSI-specific and require library-specific methods to be used.

6.4.3 Generating Designs

A file of input netlist specifications can be loaded into the design environment with the **dload** function. Netlists containing a single module can also be input from the terminal using the **%M<>** read macro. For example, in the session segment shown below I specify a netlist containing a single 4-bit adder module.

```
[4] > %M<>
> add(A^4 B^4 C.0)
>     (S^4 C.4);
<module(0):ADD.3.1.?>
```

DTAS reads the specification and generates the appropriate netlist structure, which it names **<module(0):ADD.3.1.?>**.

The portion of the name **module(0)** identifies the netlist as containing a single module. The module has id number 0, which indicates how many modules have been generated by DTAS. Thus, this is the first module to be generated in this session. The netlist can be referenced by the module's id number, e.g., **%0M**. Following the colon, the symbol **ADD** indicates that the module is of function type ADD; '**.3**' indicates that its specification matched the fourth ADD component type; '**.1**' indicates that the specification is the second to match this component type; and '**.?**' indicates that the module is uninstantiated.

The design process is initiated with the **design** function, which returns a list of netlists that are alternative designs of the input netlist. For instance,

```
[5] > (setf alts (design %0M))
(<module(0):ADD.3.1.0.0>
 <module(107):ADD.3.1.2.0>
 <module(108):ADD.3.1.1.0>)
```

generates three alternative designs for the adder module.

6.4.4 Evaluating Designs

The area and delay of a design can be examined with **show-pchars**, e.g.,

```
[6] > (show-pchars %0M)
/* USING METHOD #34 */
ADD.3.1(A^4 B^4 C.0)
       (S^4 C.4)
  :AREA  32
  :DELAY       | S[1] S[2] S[3] S[4]  C.4
         -------------------------------
         A[1]  | 1.99 3.17 4.35 5.53 5.68
         A[2]  |   -- 1.99 3.17 4.35 4.50
         A[3]  |   --   -- 1.99 3.17 3.32
         A[4]  |   --   --   -- 1.99 2.14
         B[1]  | 1.99 3.17 4.35 5.53 5.68
         B[2]  |   -- 1.99 3.17 4.35 4.50
         B[3]  |   --   -- 1.99 3.17 3.32
         B[4]  |   --   --   -- 1.99 2.14
         C.0   | 0.89 2.07 3.25 4.43 4.58
```

This function also outputs the id number of top-most decomposition method used in generating this particular implementation and a synopsis of specification.

A delay versus area graph comparing design alternatives is drawn with the **show-plot** function, e.g.,

```
[7] > (show-plot alts)
      DELAY
        |
  5.68 -|  * 0 (0%,0%)
        |
        |
        |
        |
        |
        |
  4.53 -|                                  * 107 (78%,-20%)
  4.43 -|                                        * 108 (97%,-22%)
        |
        +--+-----------------------------------+------+-- AREA
          32                                  57     63
```

This graph lists the alternatives by id number of the top-most module. It also lists the percent difference in area and delay from the smallest (leftmost) implementation. For instance, alternative 108 is 97 percent larger than alternative 0 but delay is reduced by 22 percent. Delay versus area comparison graphs can also be output with greater precision as pic files.

6.4.5 Validating Designs

The design environment supports a rudimentary simulation capability. The function **simulate** extracts set of Boolean equations from a design and outputs a function table mapping inputs to outputs. Because the complete function table for even small components, such as a 4-bit adder, is quite large, **simulate** can be run interactively, e.g.,

```
[8] > (simulate %108M :query)
X^4 ? (0 1 0 1)
Y^4 ? 5
CO ? 0
S^4 = (1 0 1 0) 10
CO = (0) 0
X^4 ? 12
Y^4 ? 10
CO ? 0
S^4 = (0 1 1 0) 6
CO = (1) 1
X^4 ? 0
Y^4 ? 0
CO ? 1
S^4 = (0 0 0 1) 1
CO = (0) 0
X^4 ? 15
Y^4 ? 15
CO ? 1
S^4 = (1 1 1 1) 15
CO = (1) 1
X^4 ?
```

where inputs can be entered as lists of binary signals or in digital form. This feature allows designs to be partially validated on selected inputs.

6.4.6 Inspecting and Outputting Designs

The hierarchical structure of designs can be examined graphically with the **inspect** function, e.g.,

```
[8] > (inspect alts)
```

The **inspect** function first outputs each design alternative to a file **module***i*.**nl** (where *i* is the id number of the design) using hierarchical, structural VHDL. It then makes a system call to the command **xdp**. (This assumes DTAS is being run under X11.) The **xdp** command opens a window for each design file, which displays each module in the design as a box and each wire as a line connecting boxes. Unless a module is

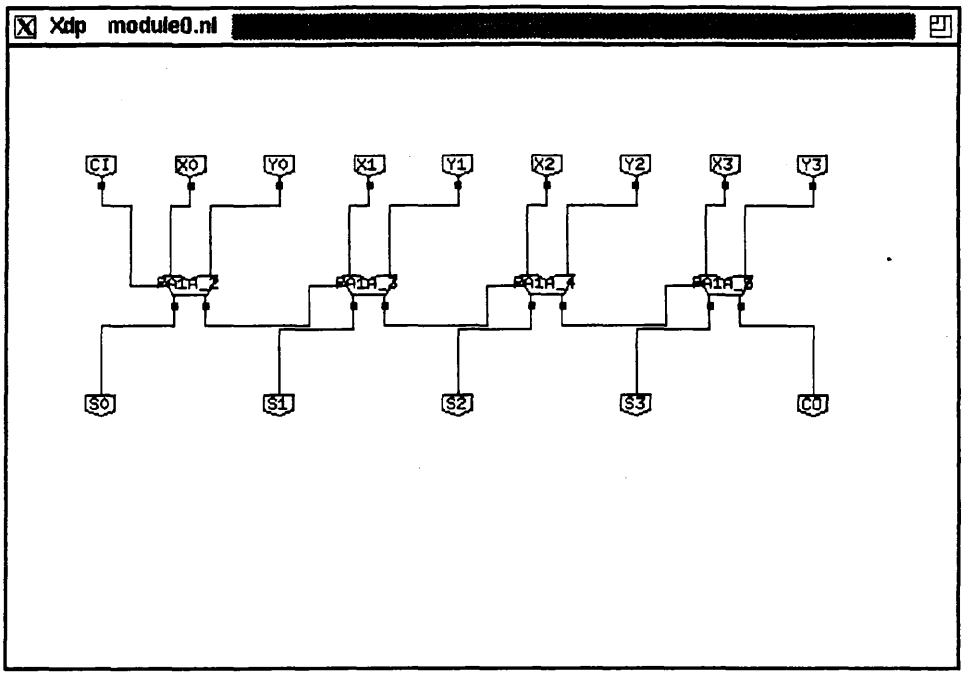


Figure 6.9: 4-bit adder using ripple-carry style and 1-bit library cells (FA1A).

implemented as a library cell, its netlist implementation can be examined by clicking the middle button on its box.

Examples for the design in `alts` are shown in Figures 6.9, 6.10, and 6.11. The design shown in Figure 6.9 depicts a ripple-carry implementation using four 1-bit library adders (FA1A), while the design shown in Figure 6.10 depicts a library-specific implementation using of a 4-bit library adder (FA4). The design shown in Figure 6.11(a) depicts a CLA implementation; Figure 6.11(b) and (c) depict the Boolean implementations of the adder and CLA components, respectively.

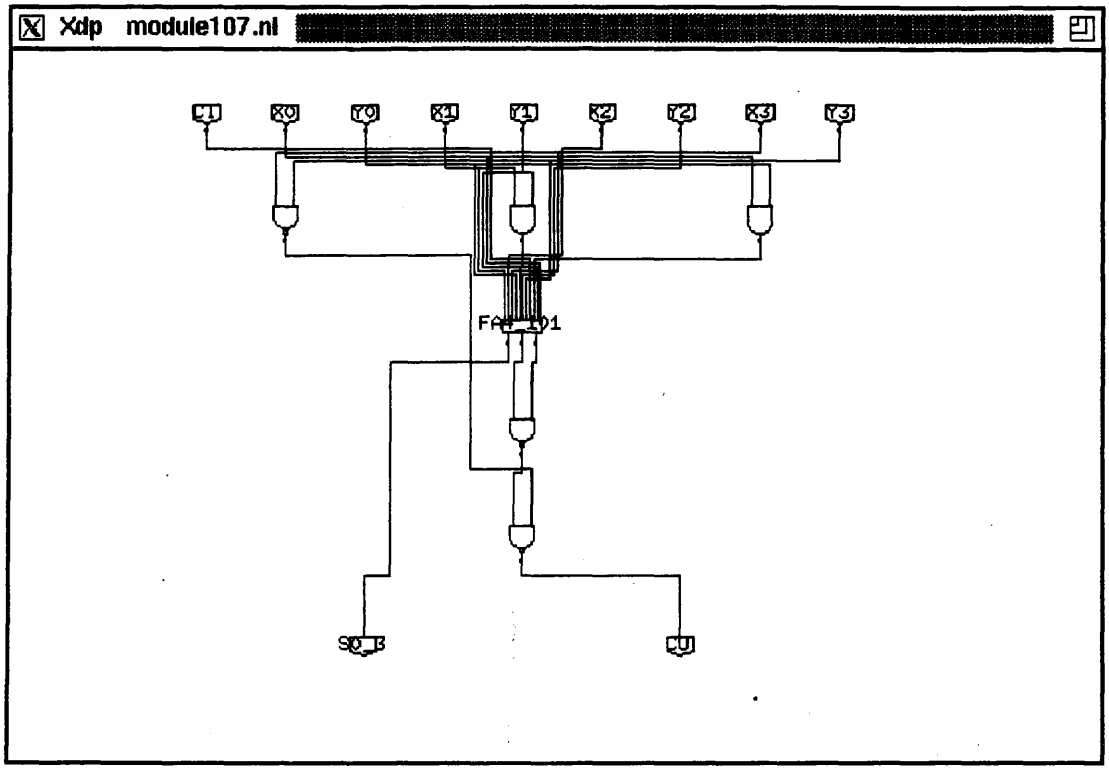
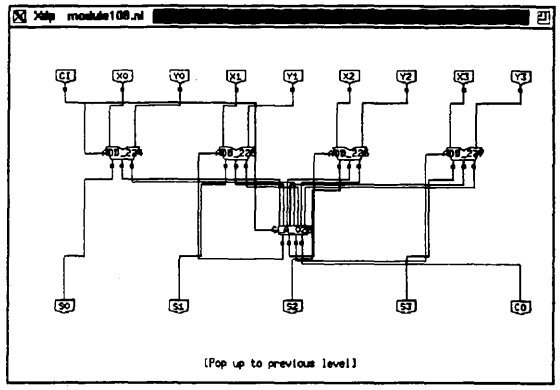
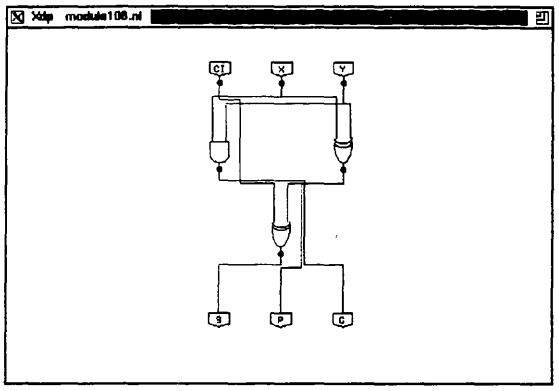


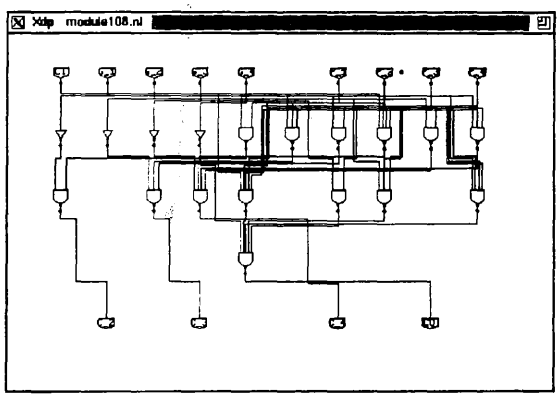
Figure 6.10: 4-bit adder using 4-bit library cells (FA4).



(a)



(b)



(c)

Figure 6.11: 4-bit adder using CLA style: (a) four 1-bit adders and CLA; (b) 1-bit adder w/carry propagate (P) and generate (G) outputs; and (c) 4-bit CLA.

Designs can also be output in one-level, structural VHDL using the **show-vhdl** function, e.g.,

```
[9] > (show-vhdl %107M)
entity ADD_3_1_2 is
  port(X1,X2,X3,X4,Y1,Y2,Y3,Y4,C0: in BIT;
       S1,S2,S3,S4,CO: out BIT);
end ADD_3_1_2;
architecture Structure_View of ADD_3_1_2 is
-- component declarations
   component INPORT
     port(I: in BIT;
          O: out BIT);
   end component;
   component OUTPORT
     port(I: in BIT;
          O: out BIT);
   end component;
   component AND_AN2
     port(I1,I2: in BIT;
          O1: out BIT);
   end component;
   component NAND_ND2
     port(I1,I2: in BIT;
          O1: out BIT);
   end component;
   component ADD_FA4
     port(A1,A2,A3,A4,B1,B2,B3,B4,C0,G0,G1: in BIT;
          S1,S2,S3,S4,C4,P3: out BIT);
   end component;
-- internal signal declarations
   signal n0,n1,n2,n3,n4,n5,n6,n7,n8,n9: BIT;
   signal n10,n11,n12,n13,n14,n15,n16,n17,n18,n19: BIT;
-- component instantiations
```

```
begin
   X1: INPORT port map (I=>X1,O=>n0);
   X2: INPORT port map (I=>X2,O=>n1);
   X3: INPORT port map (I=>X3,O=>n2);
   X4: INPORT port map (I=>X4,O=>n3);
   Y1: INPORT port map (I=>Y1,O=>n4);
   Y2: INPORT port map (I=>Y2,O=>n5);
   Y3: INPORT port map (I=>Y3,O=>n6);
   Y4: INPORT port map (I=>Y4,O=>n7);
   C0: INPORT port map (I=>C0,O=>n8);
   ADD1: ADD_FA4 port map(A1=>n0,A2=>n1,A3=>n2,A4=>n3,
                          B1=>n4,B2=>n5,B3=>n6,B4=>n7,C0=>n8,
                          G0=>n14,G1=>n15,S1=>n9,S2=>n10,
                          S3=>n11,S4=>n12,C4=>n16,P3=>n17);
   NAND2: NAND_ND2 port map(I1=>n16,I2=>n17,
                            O1=>n18);
   NAND3: NAND_ND2 port map(I1=>n3,I2=>n7,
                            O1=>n19);
   NAND4: NAND_ND2 port map(I1=>n18,I2=>n19,
                            O1=>n13);
   AND5: AND_AN2 port map(I1=>n1,I2=>n5,
                          O1=>n15);
   AND6: AND_AN2 port map(I1=>n0,I2=>n4,
                          O1=>n14);
   S1: OUTPORT port map (I=>n9,O=>S1);
   S2: OUTPORT port map (I=>n10,O=>S2);
   S3: OUTPORT port map (I=>n11,O=>S3);
   S4: OUTPORT port map (I=>n12,O=>S4);
   C0: OUTPORT port map (I=>n13,O=>C0);
end Structure_View;
```

shows the structural netlist for the implementation using a single 4-bit library adder.

6.5 Technology Independence

DTAS is intended to be a technology-independent component generation system. However, there are certain aspects of component generation and technology mapping that are problematic to generalize. These aspects concern interfacing DTAS to an ASIC cell library as well as interfacing a high-level synthesis tool to DTAS.

With regard to the former issues, there is a wide variety of function cells that any ASIC cell library can *potentially* support, even though a particular library will support only a subset of these cells. For instance, ASIC libraries can potentially support adders from 1- to 32-bits in width, even though a typical ASIC library will only support 1-, 2-, 4-, and 8-bit adders. Functional cells also can have library-specific features, not normally associated with the functional unit the cell represents.

For instance, an adder cell might have a carry enable, or there may be a particular adder cell specifically intended to be connected to a particular multiplier cell.

With regard to the latter, there is even a wider variety of functional units that can be used in high-level synthesis. The range and intended behavior of the functional units used depended on the high-level synthesis tool. There is little standardization or agreement between tools at this level.

At this time, I deal with these two interface issues by factoring them out of DTAS. In particular, DTAS is initialized to its own library of generic functional units, which cover a wide range of functionality potentials, and a set of decomposition methods for mapping any specification of a unit in this library to an implementation using generic Boolean gates. To interface DTAS to a high-level synthesis tool, DTAS must be provided with a SYN file containing a set of component types describing the range of functional units used by the high-level synthesis tool and a set of decomposition methods mapping these units into the generic functional units predefined in DTAS. To interface DTAS into a ASIC library, DTAS must be provided with a SYN file that contains component types describing functional cells that are not already represented in DTAS's generic library. More importantly, DTAS must be provided with decomposition methods that map its generic components into the available library cells. For instance, if the ASIC library contains a 2-bit adder, then DTAS must be provided with a method that decomposes n-bit adders into 2-bit adders.

The generic functional units and decomposition methods supported by DTAS are listed in Appendix A, Section A.1. An example SYN file that interfaces DTAS to the GENUS component library for high-level synthesis is listed in Appendix A, Section A.2. Example SYN files that interface DTAS to three ASIC cell libraries are listed in Appendix A, Section A.3. In Chapter 8, I present an approach to technology adaptation, which partially automates the generation of SYN files for interfacing DTAS to ASIC cell libraries.

6.6 Performance Evaluation

The effectiveness of the derivational-process model of design synthesis, as implemented in DTAS, can be evaluated in regard to proficiency, completeness, coverage, robustness, and reliability. DTAS performs favorably along four of these dimensions; namely, proficiency, completeness, coverage, and reliability.

- **Proficiency**. Proficiency is a measure of the degree to which synthesized designs compare in quality to designs generated by experienced human designers.

  DTAS is proficient. Because its decomposition methods are based on human design techniques, DTAS should be able to generate designs that are similar to those generated by human designers. Because DTAS explores design alternatives, it should actually generate a superset of the designs that an experienced designer would generate. When holes are detected in its design knowledge, new methods can be added that eliminate these deficiencies.

- **Completeness**. Completeness is a measure of the degree to which the design system can synthesize possible input specifications.

  DTAS is complete. It contains decomposition methods for most of the combinatorial and sequential component classes listed in the GENUS component library (Dutt, 1988). These methods will, at the very least, decompose any specification for a GENUS component to the level of Boolean gates. The GENUS library contains 13 classes of combinatorial components, including Boolean gates, multiplexer, selector, decoder, encoder, function generator, comparator, adder/subtractor, ALU, multiplier, divider, shifter, and barrel shifter, and 5 classes of sequential components, including register, counter, register file, stack, and memory.

- **Coverage**. Coverage is a measure of the degree to which the design system is able to take advantage of cells available in the given ASIC library.

  DTAS has good coverage, to the extent that its methods decompose components into available library cells.

- **Reliability**. Robustness is a measure of the degree to which the design system generates designs in a timely manner using "reasonable" computing and human resources.

  DTAS is reliable. Although I have not investigated the upper bounds on the time and space complexity of the design process, I have run DTAS on several large (> 32-bit data path) examples. If DTAS were to require a full 24 hours to synthesize a large specification, this would be acceptable. In its current Common Lisp implementation and running on a SUN 3/60, DTAS can synthesize a 64-to-128 multiplier in under 100 seconds. It seems likely that it DTAS can handle larger tasks in under 24 hours.

The dimension along which DTAS performs poorly is robustness. Robustness is a measure of the degree to which a design system can maintain the quality of its designs in the face of changes in the given cell library. As indicated above, the proficiency and the coverage of DTAS is largely a factor of its decomposition methods. If its methods are crafted for a given library, then DTAS will be proficient and have good coverage. However, as the available cells change, these methods must be recast

and new methods added to mirror the changes. This maintenance task requires a certain degree of expertise with the DTAS design language, as well as an understanding of existing methods. As a result, frequent changes to the cell library could make RT synthesis via DTAS prohibitively expensive.

To overcome this "robustness problem," I have investigated approaches to automating the task of maintaining the base of decomposition methods against library changes. The approach I have developed uses fundamental principles of logic design to generate library-specific methods on a cell-by-cell basis. I have implemented this approach in the LOLA subsystem, which I describe in Chapter 8.

Chapter 7
Validating Component Decomposition

In this chapter, I present experimental results that validate the utility of the component decomposition algorithm. First, I outline the nature of the experiments and explain how they validate the component decomposition algorithm; then, I describe the benchmark data files used in the experiments; next, I present empirical results collected from the experiments and discuss their significants; finally, I present a summary of these results. Results were collected using the DTAS component generation system as an implementation of the component decomposition algorithm. Target ASIC cell libraries include an MCNC standard cell library and a macrocell library from LSI Logic, Inc.

7.1 Experiments

In this dissertation, I claim that a symbolic pattern-matching approach to component generation, such as that defined in the component decomposition algorithm, has the following significant benefits:

1. It supports the use of complex functional library cells, such as MUXs, adders, and ALUs, in the generation of designs for generic functional components;
2. It effectively searches the design space for designs that make desirable trade-offs between design constraints, such as area and delay.

To validate these claims, I have run four sets of experiments using the DTAS component generation system as an implementation of the component decomposition algorithm.

The first set of experiments tests the ability of the component decomposition algorithm to control search, as well as the effectiveness of its search control strategy. As this first set of experiments will show, the size of the design space for even small

components is quite large. As a result, these experiments were limited to a set of small examples.

The second set of experiments tests the ability of the component decomposition algorithm to use alternative design styles, as well as its ability to find a range of designs that make desirable trade-offs between design characteristics, in particular, between area and delay and to scale up to large (> 32-bit data inputs) designs. This set of experiments was performed over a collection of benchmark GENUS component specifications; designs are mapped into an MCNC benchmark standard cell library.

The third set of experiments tests the ability of the component decomposition algorithm to generate high-quality designs. In this set of experiments, designs generated by DTAS were passed on to MISII for further optimization. A comparison of both sets of designs is presented. This experiment is run on an 8-bit adder, ALU, and multiplier. Designs are mapped into an MCNC benchmark standard cell library.

The final set of experiments tests the ability of the component decomposition algorithm to use complex functional components in its designs, as well as the effects of such components on design quality. In this set of experiments, designs are mapped into two subset of LSI Logic, Inc.'s macrocell library. The first library subset is limited to simple Boolean cells; the second includes complex functional cells (to 50 equivalent 2-input NAND gates).

7.2 The Data Set

In running the experiments, DTAS was provided with component types and decomposition methods for designing components drawn from the GENUS library. GENUS (Dutt, 1988) is a library of generic microarchitecture components for use in high-level synthesis. Input specifications are drawn from a collection of benchmark data files defining a large variety of component specifications from the GENUS library. Component specifications are written in VHDL.

7.2.1 The DTAS SYN Files

As noted in Chapter 6, component types and decomposition methods are organized in SYN files. A complete listing of the SYN files used in these experiments is given in Appendix A. This set also includes SYN files for memories (i.e., latches and flip flops, registers, and register files), shifters, and counters. These component types are not included in my experiments because the feedback loops implicit in their

implementations is problematic for the delay computation routines supported in the DTAS design environment, making resulting designs difficult to analyze and evaluate.

The SYN files provided to DTAS are organized as outlined below.

- GATES.SYN

  Defines generic Boolean gates, including the function types: AND, OR, NAND, NOR, XOR (exclusive-OR), XNOR (exclusive-NOR or equivalence), INV (inverter), BUF (buffer), and GATE (for other Boolean gates, such as wired-ORs). With the exception of INV, BUF, and GATE, all gates are defined as "bitwise" components, i.e., they have m n-bit inputs and out n-bit output. INVs and BUFs are defined as having one n-bit input and one n-bit output.

- DECODE.SYN

  Defines n-bit binary and BCD decoders and encoders.

- MUX.SYN

  Defines bitwise multiplexers (MUX), i.e., which select from m n-bit inputs that are directed to an n-bit output. The select port can be unary or binary encoded. The order in which inputs are mapped to select logic defaults to least significant to most significant, but the order can also be defined by the user.

- COMPARE.SYN

  Defines an n-bit comparator (COMPARE) that can test as many as six relations: equal to (EQ), not equal to (NEQ), greater than (GT), less than (LT), greater than or equal to (GEQ), and less than or equal to (GEQ). This component type optionally accepts an carry-equal, carry-greater than, and carry-less than inputs. There is no operator select logic; all specified operations are computed in parallel and directed to distinct 1-bit outputs.

- ADD.SYN

  Defines an n-bit adder (ADD) with optional carry input and carry enable, as well as carry output and carry propagate and generate. There are two fundamental styles of n-bit adders: *ripple carry* and *carry look-ahead.* There are also two fundamental styles of 1-bit adders: *AND/OR implemented* and *NAND implemented*; the latter is typically faster in CMOS technologies. There is also an inverted style of adder used in combining the partial products of an array multiplier.

- CLA.SYN

  Defines n-bit carry look-ahead generators (CLA) and carry propagate and generate generators (PG). For $n > 4$, CLAs are decomposed into a series of $\frac{n}{4}$-bit CLAs. For $n \leq 4$, CLAs and PGs there are two styles: *AND/OR implemented* and *NAND implemented*; the latter is typically faster in CMOS technologies.

- MULT.SYN

  Defines an n-by-m multiplier (MULT), i.e., where one input is n-bits and the other is m-bits; the single output has width $n + m$. There are two styles of multipliers: *matrix* (or *array*) and *Wallace tree* (or *tree*). This SYN file also defines a type of 4-input carry-save (CSA) adder for use in its definition of a tree multiplier.

  An n-by-n tree multiplier consists of four $\frac{n}{2}$-by-$\frac{n}{2}$ multipliers (any style), which compute four partial products, a CSA adder, which computes the sum and carry for each of three partial products, and a n-bit adder (any style), which combines the sum and carry outputs of the CSA adder.

- ALU.SYN

  Defines an n-bit arithmetic logic unit (ALU) with optional carry input, carry output, carry propagate and generate, and comparison output. The ALU can compute as many as four arithmetic operations: ADD, SUB, INC (increment by one), DEC (decrement by one); eight comparison operations: EQ, NEQ, GT, LT, GEQ, GEQ, ZEROP (equal to zero), and ONEP (equal to one); and 16 logic operations: AND, OR, NAND, NOR, XOR, XNOR, LID and RID (left or right input), LNOT and RNOT (left or right input inverted), LINHI and RINHI (left inhibits right or vice versa), LIMPL and RIMPL (left implies right or vice versa).

  There are two styles of ALU: *integrated* and *segregated.* With the integrated style, all operations are computed by placing external logic on the inputs and outputs of an adder with carry enable. With the segregated style, arithmetic and comparison operations are separated from logic operations. The arithmetic and comparison operations are computed by placing external logic on an adder *without* a carry enable, while the logic operations are computed with a function generator. The appropriate outputs are selected with a 2-input, n-bit MUX. The operator select port can be unary or binary encoded. An example of these two styles was seen earlier in Chapter 3, Section 3.5.

- SHFT.SYN

  Defines an n-bit shifter that can shift its inputs by m-bits to the left or right (for $m < n$). A shifter has four operations: shift left, shift right, pass inputs, pass zeros.

- MEMORY.SYN

  Defines flip flops and n-bit registers. There are four styles of flip flops: *D*, *T*, *JK*, and *SR*, each of which can optionally have a asynchronous set and reset inputs; a D-style flip flop can also have an asynchronous load.

  Registers are defined in terms of D flip flops. A register can optionally have asynchronous or synchronous load, set, and reset inputs.

- CNTR.SYN

  Defines an n-bit up/down counter with optional asynchronous clear.

7.2.2 GENUS Data Files

Input specifications for the experimental results presented here were drawn from a collection of GENUS data files. Each data file contains several specifications of the GENUS component designated by the file's name; e.g., the file `mult.nl` contains specifications of a GENUS multiplier. The difference between the specifications in a file is the width of their data inputs, which typically varies from 4 to 64 bits. Most GENUS components have additional parameters that can effect performance; variations of these parameters are specified in separate files. Components are specified using structural VHDL.

7.2.3 Sample Cell Libraries

In the experiments described below, DTAS mapped designs into three layout cell libraries:

1. The MCNC standard cell benchmark library "lib2.mis2lib" (Lisanke, 1988). This library contains three different inverters, 2-, 3-, and 4-input NAND and NOR gates, 2-input XOR and XNOR gates, 16 different two-level Boolean gates.
2. A subset of LSI Logic, Inc.'s macrocell library consisting of Boolean cells only. This library is drawn from the cells catalogued in a databook of 2-micron cells (LSI, 1987).
3. A subset of LSI Logic, Inc.'s macrocell library consisting of Boolean cells and complex functional cells, including two 1-bit adder cells (FA1 and FA1A), a 4-bit adder (FA4) and two related carry look-ahead generators (CLA1 and CLA2), and 2-to-1, 4-to-2, and 8-to-4 multiplexers. This library is also drawn from the cells catalogued in a databook of 2-micron cells (LSI, 1987).

A complete listing of the cells in these three libraries appears in Appendix B.

7.3 Experimental Results

In this section, I present the results for four sets of experiments examing various aspects of the component decomposition algorithm as implemented in DTAS.

1. *The Effectiveness of Search Control*, which compares the the designs generated by DTAS with and without the use of search control.
2. *Finding Desirable Design Alternatives*, which shows how DTAS can find compare and alternative design styles to find designs that make favorable trade-offs between area and delay.
3. *Comparative Design Quality*, which compares the quality of designs generated by DTAS to those generated with the MISII logic optimization system.
4. *Designing with Functional Cells*, which compares the quality of designs generated with and without the use of complex functional cells.

In each experiment, I list the example components designed, present the results on a delay versus area graph, and discuss their significants.

For each experiment, the performance filtering function used discarded all designs that did not make favorable trade-offs between area and delay. I call this function the *bounded-curve filter*, because the designs it returns mark the lower-bound curve of the design space when plotted on a delay versus area graph. The bounded-curve filter was defined earlier in Chapter 4, Section 4.5, Figure 4.13, as the filtering function SELECT-BOUNDED-CURVE.

Resulting designs are presented as points in delay versus area graphs, where delay is the maximum delay through the circuit and area is the cumulative cell area. Design points are typically labeled from 1 to n, where design 1 has the least area and design n has the smallest maximum delay. Designs 2 through n are also typically labeled by two percentages, e.g., (82%,-25%). The first number is the percentage difference in area from the smallest design, e.g., the labeled design is 82 percent larger than the smallest design. The second number is the percentage difference in delay, e.g., the labeled design is 25 percent faster than the smallest design.

Experimental results are shown with regard to three arithmetic component types: adders, 8- and 16-function ALUs, and multipliers. These component types were selected because for two reasons. First, they are typical of components found on the critical path during datapath synthesis, and, second, they are markedly effected by style selection.

7.3.1 The Effectiveness of Search Control

As explained in Chapter 4, Section 4.5, the component decomposition algorithm uses two principles to control the size of the design space it searches. The first principle says to ignores netlist implementations containing two or more modules defined by the same component specification but that are not instances of the same component implementation. The second principle says to apply a user-defined performance filtering function to lists of alternative implementations of the same component, discarding all implementations not returned as "acceptable" by the filtering function.

In this set of experiments, I examined the effectiveness of this search control strategy. In particular, I compared the designs generated when no search control was applied with the designs generated when both search control principles were applied, individually and together. Experiments were run on the following examples:

1. 1-, 4-, and 8-bit full adders;
2. 4-bit/8-function ALU and 8-bit/8-function ALUs;
3. 4- and 8-bit multipliers.

Designs were mapped into the second subset of LSI Logic's macrocell library, which contains complex functional cells.

Example 1. Design results for the 1-bit full adder are shown in Figure 7.1. As depicted in Figure 7.1(a), DTAS generated four distinct full adders from the given subset of LSI Logic's cell library. Design 1 was one of the full adder library cell (FA1A); design 2 was a Boolean implementation using NAND gates; design 3 was the other adder cell (FA1); and design 4 was another Boolean implementation using AND and OR gates.

This design space was not affected by applying the first search control principle. However, it was affected by applying the bounded-curve filtering function, which discarded all but the two adder cell designs, as depicted in Figures 7.1(b); applying both principles resulted in the same two designs being generated.

Results for the 4-bit full adder are shown in Figure 7.2. The full design space, consisting of 9329 alternative designs, is depicted in Figure 7.2(a). Application of the first search control principle reduced this space to 341 designs, as depicted in Figure 7.2(b), while using the filtering function reduced it further to three designs, as depicted in Figure 7.2(c). Using the full search control strategy also reduced the total number of designs to three, as depicted in Figure 7.2(d), but not to the exact same three as in Figure 7.2(c).

Comparing Figures 7.2(a) and (d), one can see that, by using the full search control strategy (Figure 7.2(d)), DTAS was able to find the smallest design (1), and

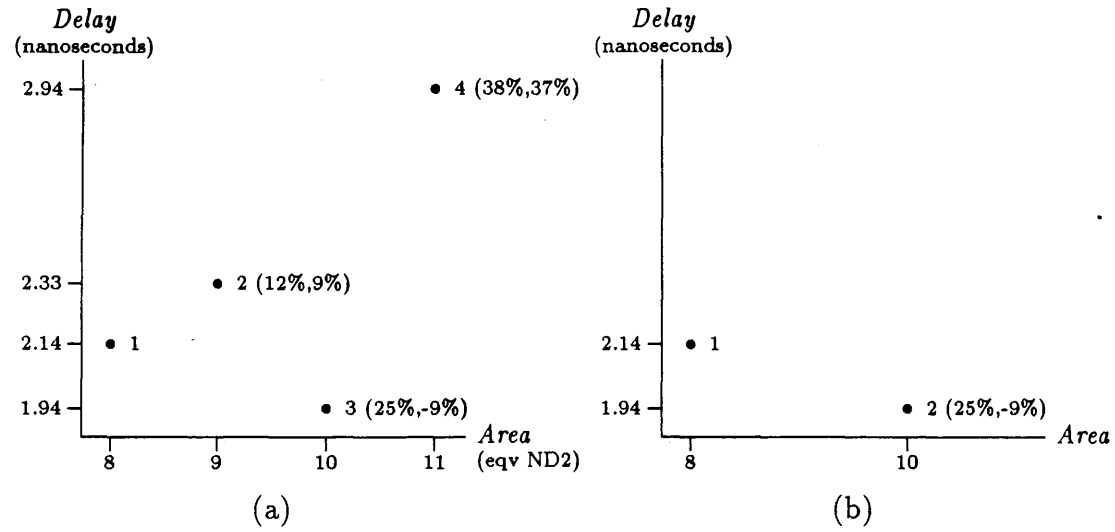


Figure 7.1: Experiment 1 – results for 1-bit adder: (a) full design space; and (b) results after applying search control.

one of several equally fast designs (3) and one of the designs that make a favorable trade-off between area and delay (2). Design 1 was a ripple-carry adder implemented with four FA1As; design 2 was implemented with the 4-bit adder cell FA4; and design 3 was a carry look-ahead adder implemented with Boolean gates.

There were clearly several other designs in Figure 7.2(a) that made "more favorable" trade-offs between area and delay than design 2 of Figure 7.2(d). The fact that these designs did not appear in Figure 7.2(b) indicates that they used combinations of implementations for identically specified modules in the same netlist. In particular, the six designs appearing in the lower left-hand corner of Figure 7.2(a) were ripple-carry adders using various combinations of the two 1-bit adder cells (FA1A and FA1) and the NAND-implemented adder. (Although the NAND adder has a worse maximum delay from summands to sum output, it has a short carry delay.) Using only the performance filtering function, DTAS was able to find one of these designs, shown as design 2 in Figure 7.2(c), which ripples one FA1 followed by three FA1As; the NAND adder was discarded by the filtering function, as presented earlier in Figure 7.1(b).

Although from this one example it would seem that a better mix of designs can be generated *without* the first search control principle, this is computationally infeasible. Scaling up to a 16-bit adder, there would be more than 2^{16} alternative

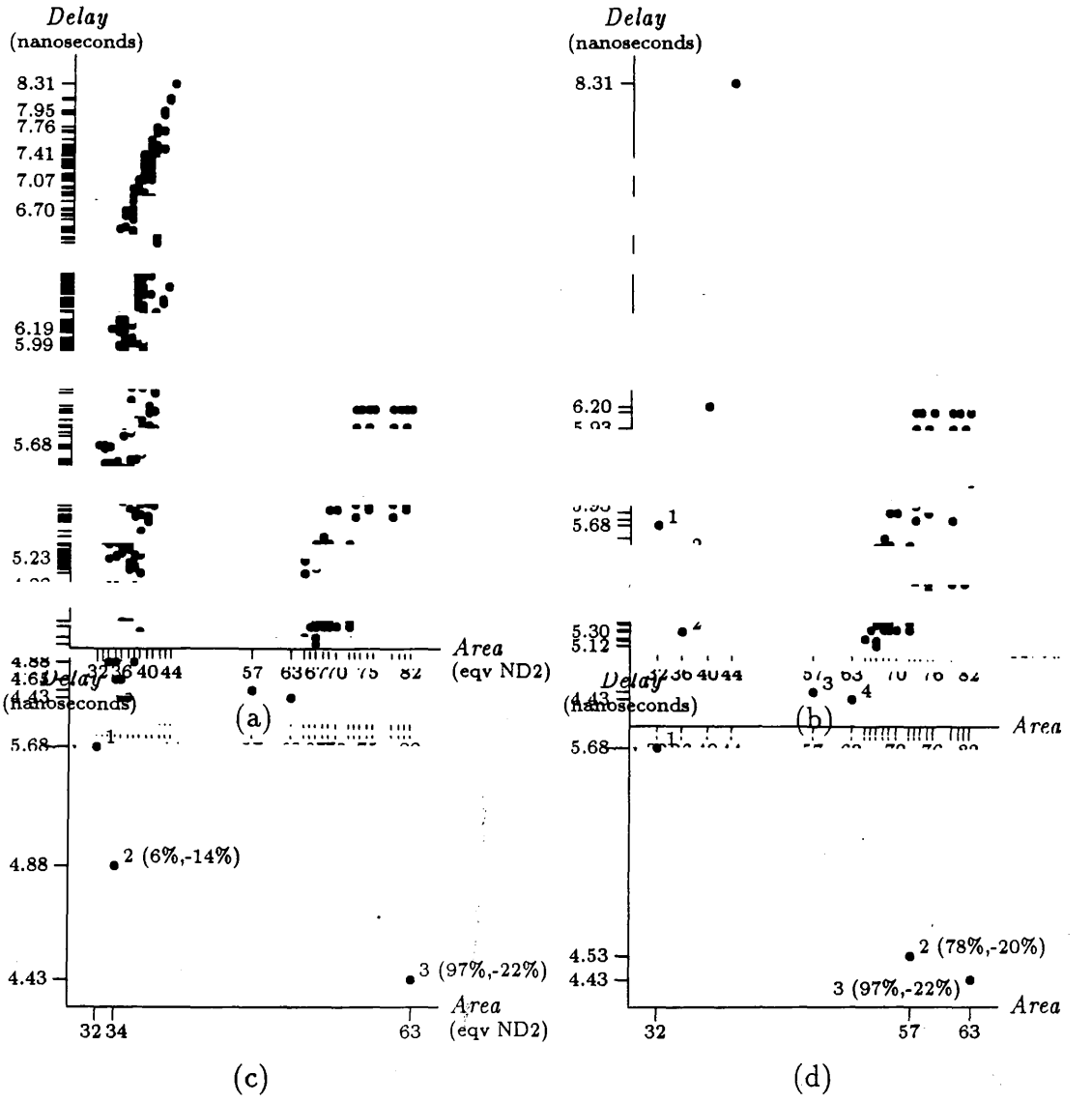


Figure 7.2: Experiment 1 – results for 4-bit adder: (a) full design space; and (b) results after applying first control principle only; (c) results after applying filtering function only; and (d) results after applying full search control.

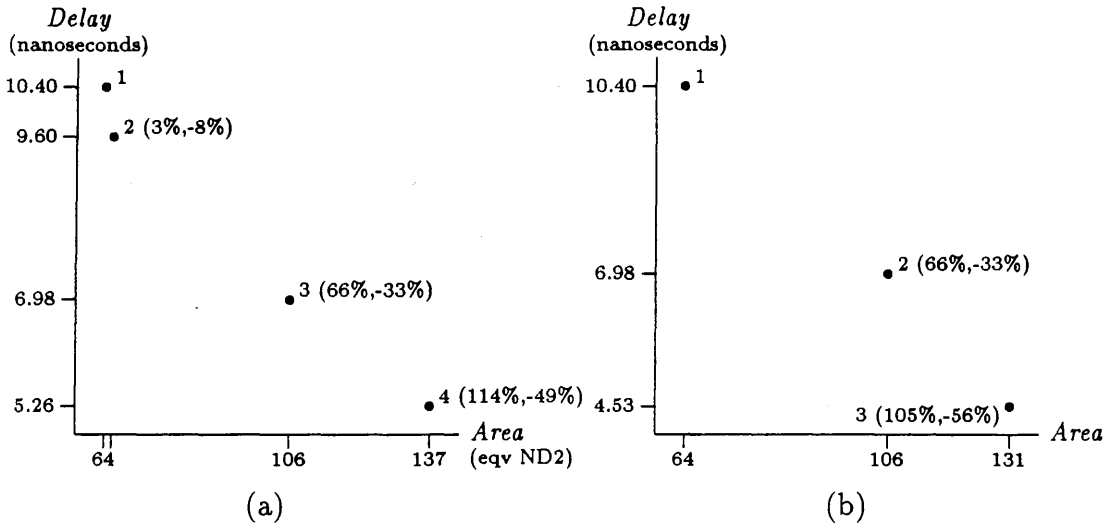


Figure 7.3: Experiment 1 – results for 8-bit adder: (a) results after applying filtering function only; and (b) results after applying full search control.

designs to evaluate and compare; scaling up further to a more realistic size of 32 bits, there would be more than 2^{32} alternative designs. Given that the first search control principle must be applied, applying both principle compares well, since three of the four best designs from Figure 7.2(b) also appear in Figure 7.2(d).

With regard to runtime performance, the complete design space (Figure 7.2(a)) was generated by DTAS in one hour and twenty two minutes, on a Spark-2 workstation; the designs resulting from the application of the full search control strategy (Figure 7.2(d)) were generated by DTAS in under 1 second.

Finally, results for an 8-bit full adder are shown in Figure 7.3. For this example, DTAS was unable to generate the complete design space or to generate designs by applying the first control principle only. The designs depicted in Figure 7.3(a) were generated using the filtering function only; the designs depicted in Figure 7.3(b) were generated using full search control.

In this case, by using the full search control strategy DTAS was actually able to find a design that was smaller and faster than the fastest design generated when using the filtering function only. This resulted because design 3 of Figure 7.3(b) used design 2 of Figure 7.2(d) (i.e., the 4-bit adder cell, FA4) as a subcomponent of the 8-bit adder. Design 2 of Figure 7.2(d) was discarded by DTAS when it used the filtering function only, so it was unavailable to be used as a subcomponent in design 3 of Figure 7.3(b).

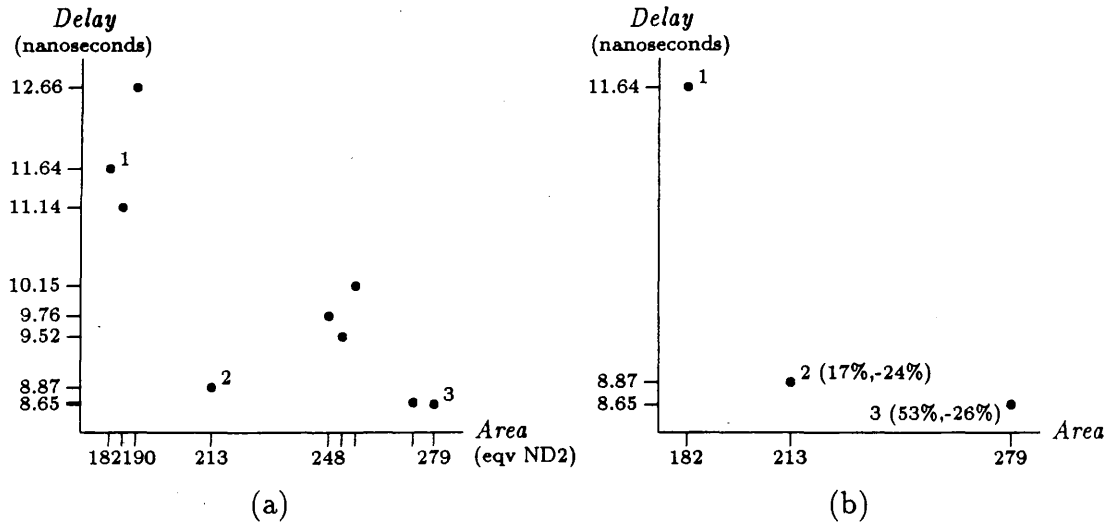


Figure 7.4: Experiment 1 – results for 4-bit/8-function ALU: (a) results after applying first control principle only; and (b) results after applying filtering function and after applying full search control.

Example 2. Results for the 4-bit/8-function ALU are shown in Figure 7.4. DTAS was unable to generate the complete design space for this example. When only the first search control principle was applied, DTAS generated 87040 alternative designs, only a few of which are depicted in Figure 7.4(a). DTAS required 11 hours and 47 minutes to generate these designs on a Spark-2 workstation. When the filtering function was applied only, as well as when the full search control strategy was applied, DTAS generated the same three designs, which are depicted in Figure 7.4(b). DTAS generated these designs in under 10 seconds.

Results for the 8-bit/8-function ALU are shown in Figure 7.5. For this example, DTAS was unable to generate the complete design space or to generate designs by applying the first control principle only. When only the filtering function was applied, DTAS generated the three designs depicted in Figure 7.5(a). When the full search control strategy was used, DTAS generated the three designs depicted in Figure 7.5(b). In the first case, DTAS generated its designs in a little over a minute; in the second case, DTAS generated its designs in under 30 seconds. This example shows another instance when DTAS was able to find a design using the full search strategy that was smaller and faster than the fastest design found using the filtering function only.

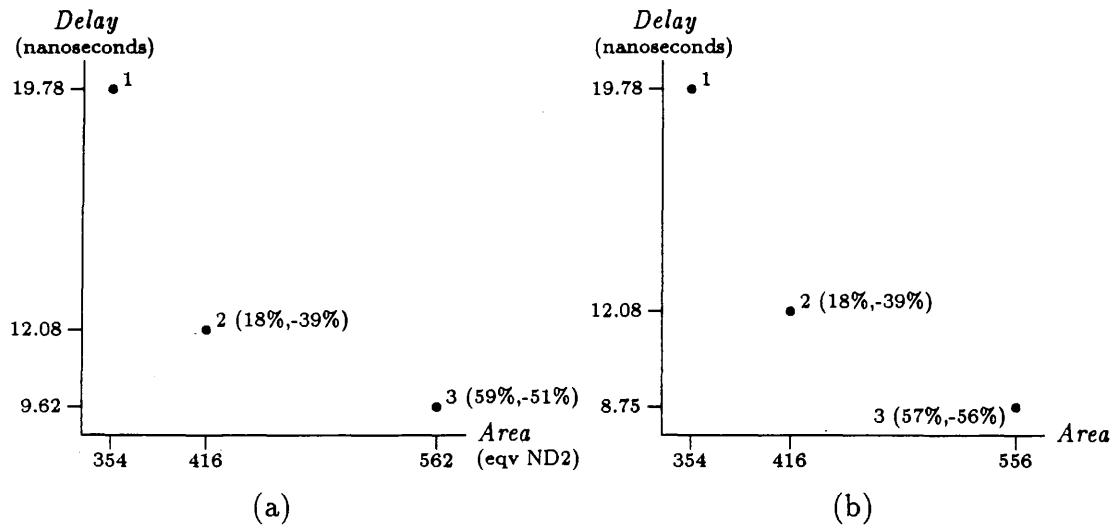


Figure 7.5: Experiment 1 – results for 8-bit/8-function ALU: (a) results after applying filtering function only; and (b) results after applying full search control.

Example 3. Design results for the 4-bit multiplier are shown in Figure 7.6. For this example, DTAS was again unable to generate either the complete design space or designs when only the first search control principle was applied. When only the filtering function was applied, DTAS generated the four designs depicted in Figure 7.6(a). When the full search control strategy was applied, DTAS generated the four designs depicted in Figure 7.6(b). In both cases, DTAS required under 7 seconds to generate its designs. As indicated in Figure 7.6, the designs generated when only the filter function was applied were not significantly better than the designs generated under full search control.

7.3.2 Finding Desirable Design Alternatives

Decomposition methods can be viewed as encapsulating design styles. Design alternatives are established when two or more methods are applicable to the same component specification. The component decomposition algorithm is capable of expanding and combining design alternatives, with the result of generate designs that make desirable trade-offs between design characteristics, such as area and delay.

Examples of this capability are apparent in the previous set of experiments. In this set of experiments, I examine this aspect of the component decomposition algorithm further. In particular, I show how this capability scales up from the small

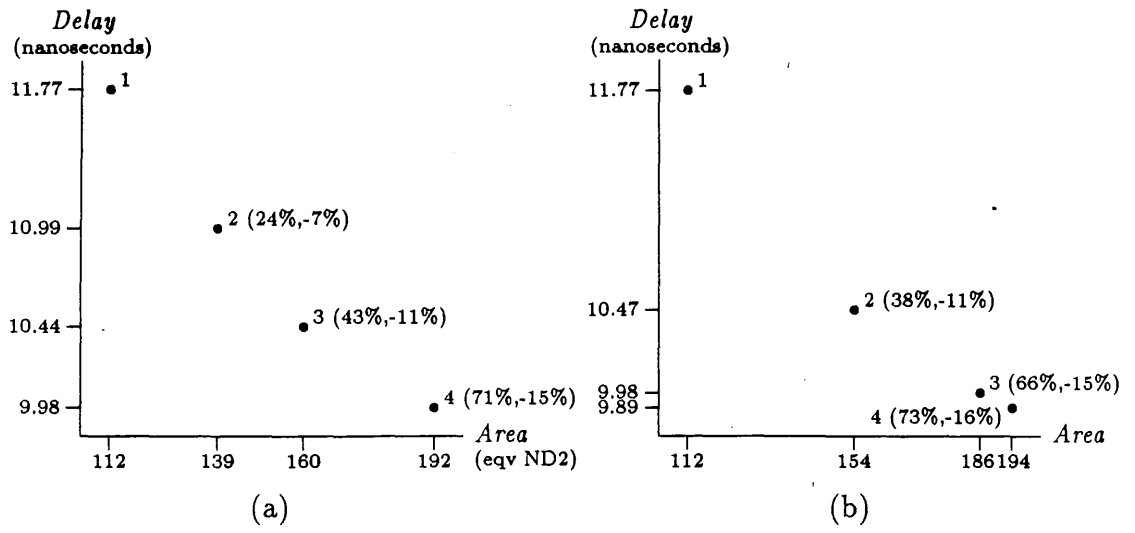


Figure 7.6: Experiment 1 – results for 4-bit multiplier: (a) results after applying first control principle only; and (b) results after applying filtering function and after applying full search control.

examples seen above to large examples, i.e., with data widths greater than 32 bits. These experiments include the following examples:

1. 4-, 8-, 16-, 32-, and 64-bit adders;
2. 4-, 8-, 16-, 32-, and 64-bit ALUs that compute 8 and 16 functions;
3. 4-, 8-, 16-, 32-, and 64-bit multipliers.

Designs were mapped into the MCNC benchmark cell library. Results for each size component are shown on the same delay versus area graphs; alternative designs for the same input specification are connected by dashed lines. Elapsed wall-clock design time is also listed.

Example 1. Design results for the 4-, 8-, 16-, 32-, and 64-bit adders are shown in Figure 7.7. DTAS has two dominant designs styles for adders: *ripple carry* and *carry look-ahead.* These two styles can be mixed, such that it is possible to ripple across carry look-ahead adders.

Only two alternative designs were generated for each of the example adders. In each case, design 1 was a full ripple-carry adder, while design 2 was a full carry look-ahead adder that uses 4-bit carry look-ahead generators. As indicated in Figure 7.7, the carry look-ahead adder was consistently almost twice the size of the ripple-carry

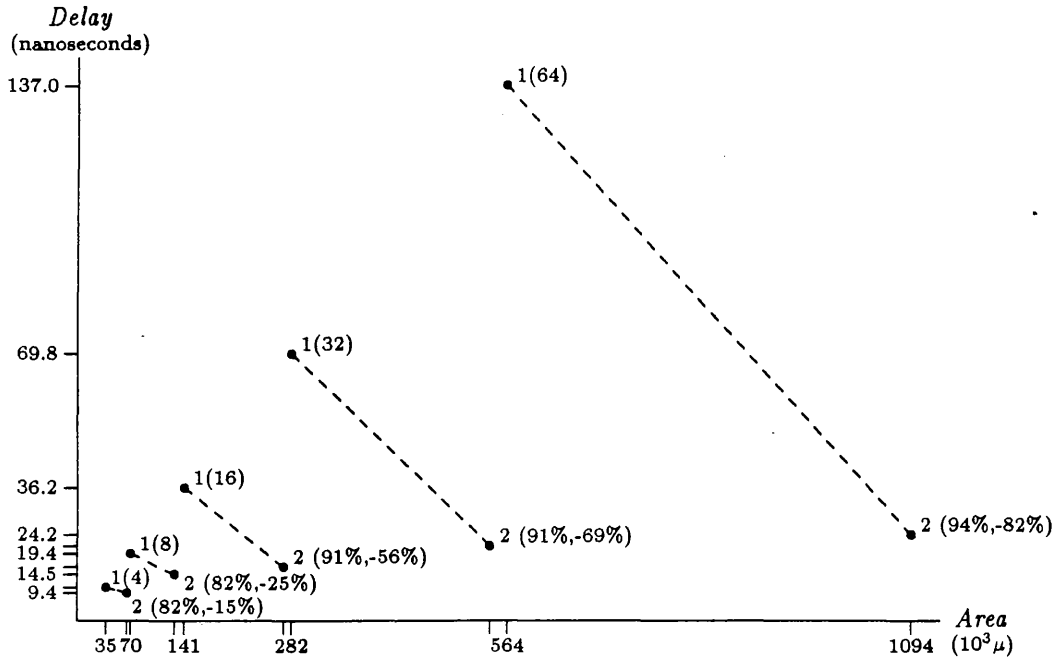


Figure 7.7: Experiment 2 – 4-, 8-, 16-, 32-, and 64-bit adders.

adder. However, as the data width increased there was also a significant percentage increase in the reduction in delay.

DTAS generated designs for the 4-bit adder in 4 seconds of wall-clock time, the 8-bit adder in 8 seconds, the 16-bit adder in 20 seconds, and the 32-bit adder in 38 seconds, and the 64-bit adder in 85 seconds.

Example 2. Design results for the 4-, 8-, 16, 32-, and 64-bit/8-function ALUs are shown in Figure 7.8; the functions computed include four arithmetic operations: addition, subtraction, increment, and decrement, and four logic operations: AND, OR, NAND, and NOR. DTAS can use two design styles for ALUs: *integrated*, in which all operations are computed through an adder with carry enable, and *segregated*, in which arithmetic and comparison operations are computed through an adder without carry enable and logic operations are computed through a function generator. An example of this style was described earlier in Chapter 3, Section 3.5. The adder subcomponent of an ALU can be any style.

For this example, three design alternatives were generated for all but the 64-bit ALU. In each case, design 1 used an integrated style and a ripple-carry adder; design 2 used an integrated style and a carry look-ahead adder; and design 3 used a

Figure 7.8: Experiment 2 – 4-, 8-, 16-, 32-, and 64-bit/8-function ALUs.

segregated style and a carry look-ahead adder. As indicated in Figure 7.8, the mix of integrated ALU and carry look-ahead adder was consistently less than 15 percent larger than the smallest design and less than 4 percent slower than the fastest design; at 64-bits this mix is actually the fastest. While the mix of styles used in design 2 did not necessarily result in either the smallest or fastest designs, it did result in designs that made a very favorable trade-off between area and delay.

DTAS generated designs for the 4-bit ALU in 8 seconds of wall-clock time, the 8-bit ALU in 17 seconds, the 16-bit ALU in 35 seconds, and the 32-bit ALU in 1 minute and 52 seconds, and the 64-bit ALU in under 6 minutes.

Results for the 16-function ALUs are shown in Figure 7.9. The functions computed include four arithmetic operations: addition, subtraction, increment, and decrement, four comparison operations: equal, less than, greater than, and equals zero, and eight logic operations: AND, OR, NAND, NOR, XOR, XNOR, INV (leftmost input), and IMPLICATION (leftmost input).

These results are similar to those shown in Figure 7.8. Three designs were generated for each of the ALUs, using the same three mix of design styles. Again, while the mix of design styles found in design 2 resulted in generating neither the

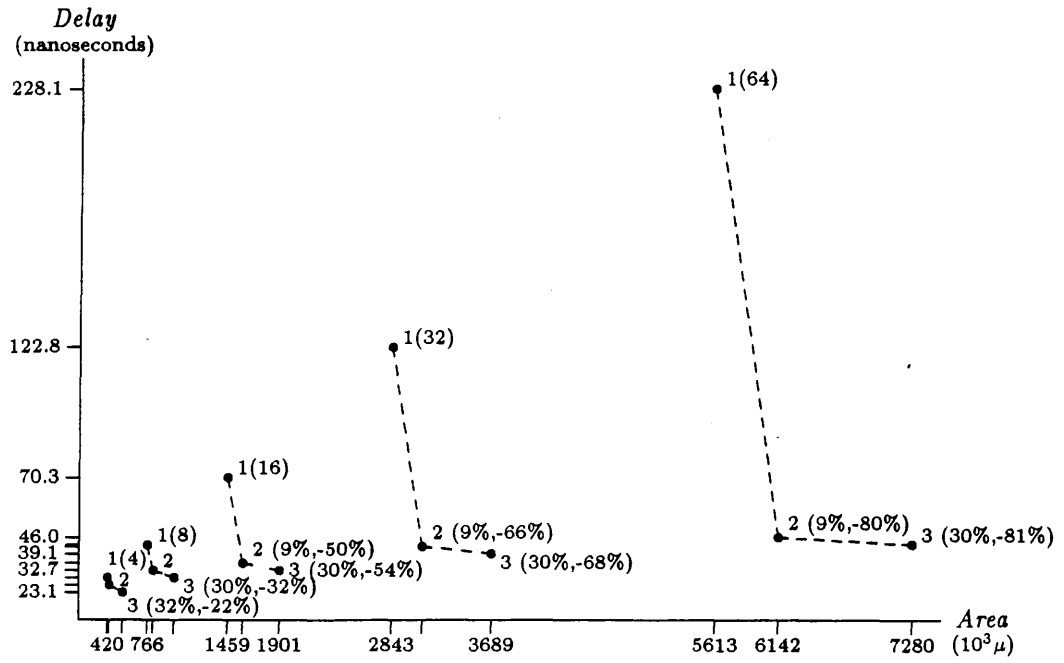


Figure 7.9: Experiment 2 – 4-, 8-, 16-, 32-, and 64-bit/16-function ALUs.

smallest or fastest designs, this mix did make very favorable trade-offs between area and delay; even more favorable in this case than in the former.

DTAS generated designs for the 4-bit ALU in 10 seconds of wall-clock time, the 8-bit ALU in 22 seconds, the 16-bit ALU in 55 seconds, and the 32-bit ALU in 2 minutes and 20 seconds, and the 64-bit ALU in under 8 minutes.

Example 3. Design results for the 4-, 8-, 16-, 32-, and 64-bit multipliers are shown in Figure 7.10. As with adders, there are two dominant styles for multipliers: *matrix* (or *array*) and *Wallace tree* (or *tree*). An n-bit Wallace-tree multiplier consists of four $\frac{n}{2}$-bit multipliers, which can be either style, plus an n-bit adder, which can be any adder style. Thus, it is possible to generate a number of hybrid styles for multipliers.

As depicted in Figure 7.10, a wide range of designs were generated for each multiplier specification. In each case, design 1 was always a full matrix multiplier, and the last design was always a full tree multiplier with carry look-ahead adders in each tree. Design 2 was typically a tree of four matrix multipliers and a ripple-carry adder, design 3 was typically a tree of matrix multiplies and a carry look-ahead adder. Other designs use a tree of tree of matrix style with carry look-ahead adders at the

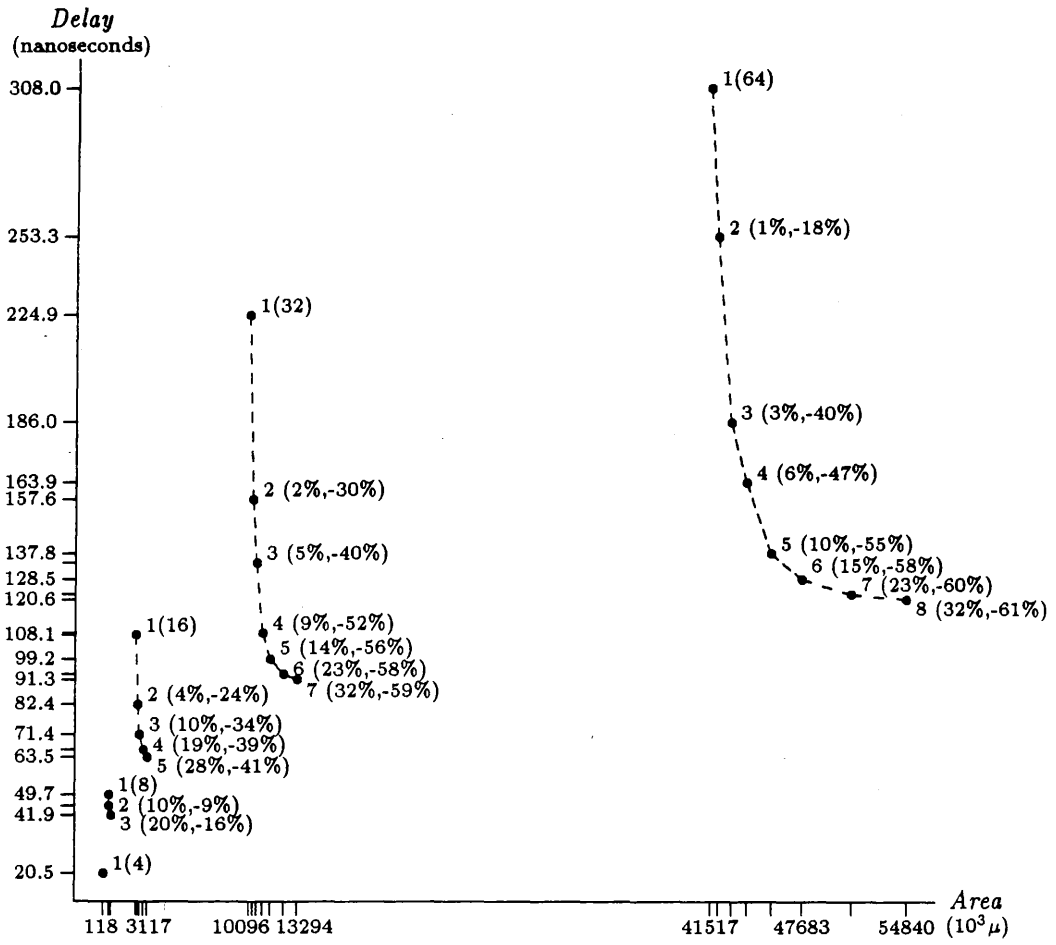


Figure 7.10: Experiment 2 – 4-, 8-, 16-, 32-, and 64-bit multipliers.

highest level. These results again show how the ability to mix design styles allows DTAS to generate a range of designs that make desirable trade-offs between area and delay.

DTAS generated designs for the 4-bit multiplier in 3 seconds of wall-clock time, the 8-bit multiplier in 20 seconds, the 16-bit multiplier in 8 minutes, and the 32-bit multiplier in 1 hour and 22 minutes, and the 64-bit multiplier in 5 hours and 49 minutes.

7.3.3 Comparative Design Quality

The component decomposition algorithm is able to generate a range of designs that make desirable trade-offs between area and delay, but this is no indication of overall design quality. Because it uses the same design styles (as encapsulated in decomposition methods) as applied by human designers, the component decomposition algorithm should be capable of generating designs that are close to human quality; although much more quickly and in a wider range of forms. (This assumes the availability of appropriate decomposition methods.) However, the component decomposition algorithm uses little Boolean minimization and no logic optimization, so it is of interest to compare its designs to current approaches to component generation based on logic synthesis.

In this set of experiments, I compare designs generated by the component decomposition algorithm with designs generated by a popular and highly-regarded logic synthesis tool, MISII (Detjens et al., 1987). These experiments include the following examples:

1. an 8-bit adder;
2. 8-bit ALUs that compute 8 and 16 functions;
3. an 8-bit multiplier;

Designs were mapped into the MCNC benchmark cell library.

MISII was used in the following manner. First, DTAS was applied to generate sets of alternative designs for each example component specification. Because I used the MCNC library, DTAS's designs were all decomposed to the level of Boolean gates. Second, for each DTAS design, the set of Boolean equations describing that design were extracted and output using Berkeley EQN format. Third, for each such EQN file, MISII was executed in batch mode, in which it input the MCNC cell library, input the EQN file, mapped the given Boolean equations into the cell library, and output the resulting design in BDNET format. Finally, DTAS loaded the BDNET design. In this way, I replicated the current approach to component generation, using DTAS as the role of a module generator that produces Boolean logic.

For each example, results for the DTAS- and MISII-generated designs are shown together in two delay versus area graphs. In the first graph, DTAS designs are compared to each other and MISII designs are compared to each other. In the second graph, each DTAS design is compared to the corresponding design generated by MISII.

When optimizing circuits using the MCNC cell library, MISII assumes a nominal drive on the inputs of 1.98 nanoseconds from low to high and 1.82 nanoseconds

from high to low; it also assumes a nominal load on the outputs of 0.10. In the comparison of results presented here, these drive and load factors were accounted for when computing maximum delay. As a result, the DTAS designs depicted in these experiments will have longer delays than the corresponding designs depicted in the last set of experiments.

Example 1. Design results for the 8-bit adder are shown in Figure 7.11. DTAS generated two alternative designs: design 1 was a ripple-carry adder, and design 2 was a full carry look-ahead adder. When the Boolean equations describing these two designs were passed through MISII, the results returned were mixed.

As depicted in Figure 7.11(a), in both cases the ripple-carry adder had the least area while the carry look-ahead adder had the shortest maximum delay. For MISII, the trade-off between area and delay were more equally proportioned than for the DTAS designs. However, this was due to the fact that MISII actually generated a worse design for the ripple-carry adder than did DTAS. As depicted in Figure 7.11(b), MISII was able to improve the design of DTAS's carry look-ahead adder by 5 percent with regard to area and 9 percent with regard to delay. However, the design generated by MISII for the ripple-carry adder was 21 percent larger and 13 percent slower.

These results indicate two things. First, the quality of designs generated by MISII is dependent upon on the given Boolean description. As illustrated here, given two alternative, yet functionally equivalent, Boolean descriptions of an 8-bit adder, MISII generated two alternative designs. Thus, DTAS's ability to find alternative designs can be a significant factor in ultimately determining design quality. Second, MISII's optimization strategy is not necessarily appropriate for regular-structured components. For instance, DTAS's ripple-carry adder was implemented with eight NAND-implemented 1-bit adders; this adder has a short delay from carry input to carry output and is relatively small, so rippling eight of them generated an adder that was only 20 slower than a full carry-look ahead adder. MISII's optimization strategy is to use two-level Boolean gates whenever possible, so it never tried a series of NAND gates to implement the critical path from carry input to carry output and, thus, never discovered the DTAS-generated design.

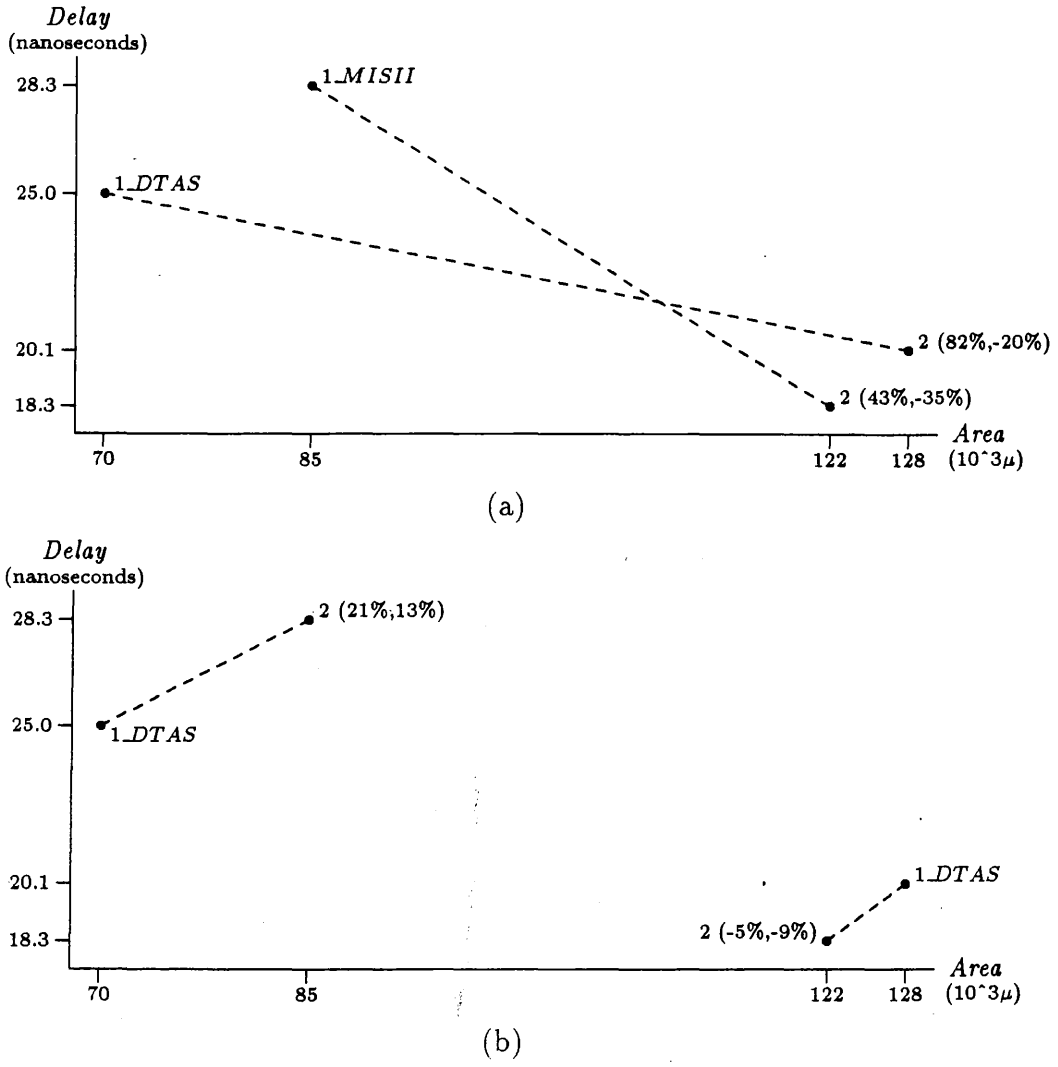


Figure 7.11: Experiment 3 – results for 8-bit adder: (a) comparing DTAS designs to each other and MISII designs to each other; and (b) comparing DTAS designs to corresponding MISII designs.

Example 2. Results for the 8-bit/8-function ALU are shown in Figure 7.12; the functions computed include four arithmetic operations: addition, subtraction, increment, and decrement, and four logic operations: AND, OR, NAND, and NOR. In this case, DTAS only generated two designs: design 1 used an integrated ALU style with a ripple-carry adder, while design 2 used an integrated ALU style with a carry look-ahead adder. In the last set of experiments (Figure 7.8), a third design alternative, using the segregated ALU style, was also generated. This design style does not appear here because it did not compare favorably to design 2 when the nominal input drive and output load were included to the computation of maximum delay.

As depicted in Figure 7.12(a), in both cases the ALU design using the ripple-carry adder had the least area while the ALU design using the carry look-ahead adder had the shortest maximum delay, which should not be too surprising given the last example (Figure 7.11). The delay versus area trade-offs were slightly different. As depicted in Figure 7.12(b), MISII was able to generate designs that were significantly smaller than those generated by DTAS, although MISII's design 1 was only 6 percent faster than DTAS's design 1 and DTAS's design 2 was actually 10 percent faster than MISII's corresponding "optimized" design.

MISII was able to reduce the area of DTAS's two designs for the following reason. As described through example in Chapter 3, Section 3.5, DTAS decomposed the ALU into an adder (with carry enable) and random combinational circuits on the inputs and outputs to compute selected operations. These circuits were described in terms of an AND/OR/NOT implementation, which MISII is extremely good at optimizing, resulting in a significant decrease in area. MISII was not able to get a proportional reduction in delay because delay through the ALUs was still dominated by the delay through the adders, which is even worse than that shown in the last example due to the carry enable logic.

Results for the 16-function ALU are shown in Figure 7.13; the functions computed include four arithmetic operations: addition, subtraction, increment, and decrement, four comparison operations: equal, less than, greater than, and equals zero, and eight logic operations: AND, OR, NAND, NOR, XOR, XNOR, INV (leftmost input), and IMPLICATION (leftmost input). Again, designs 1 and 2 used the integrated ALU style with a ripple-carry and carry look-ahead style,respectively.

As depicted in Figures 7.13(a) and (b), the comparison results are almost identical to those generated for the 8-function ALU (Figure 7.12), for the same reasons. The results for both ALUs again reaffirm the point that MISII benefits from application of alternative design styles. Further, these examples indicate that, although DTAS's designs can benefit from optimization, they are still relatively close in performance to the optimized designs.

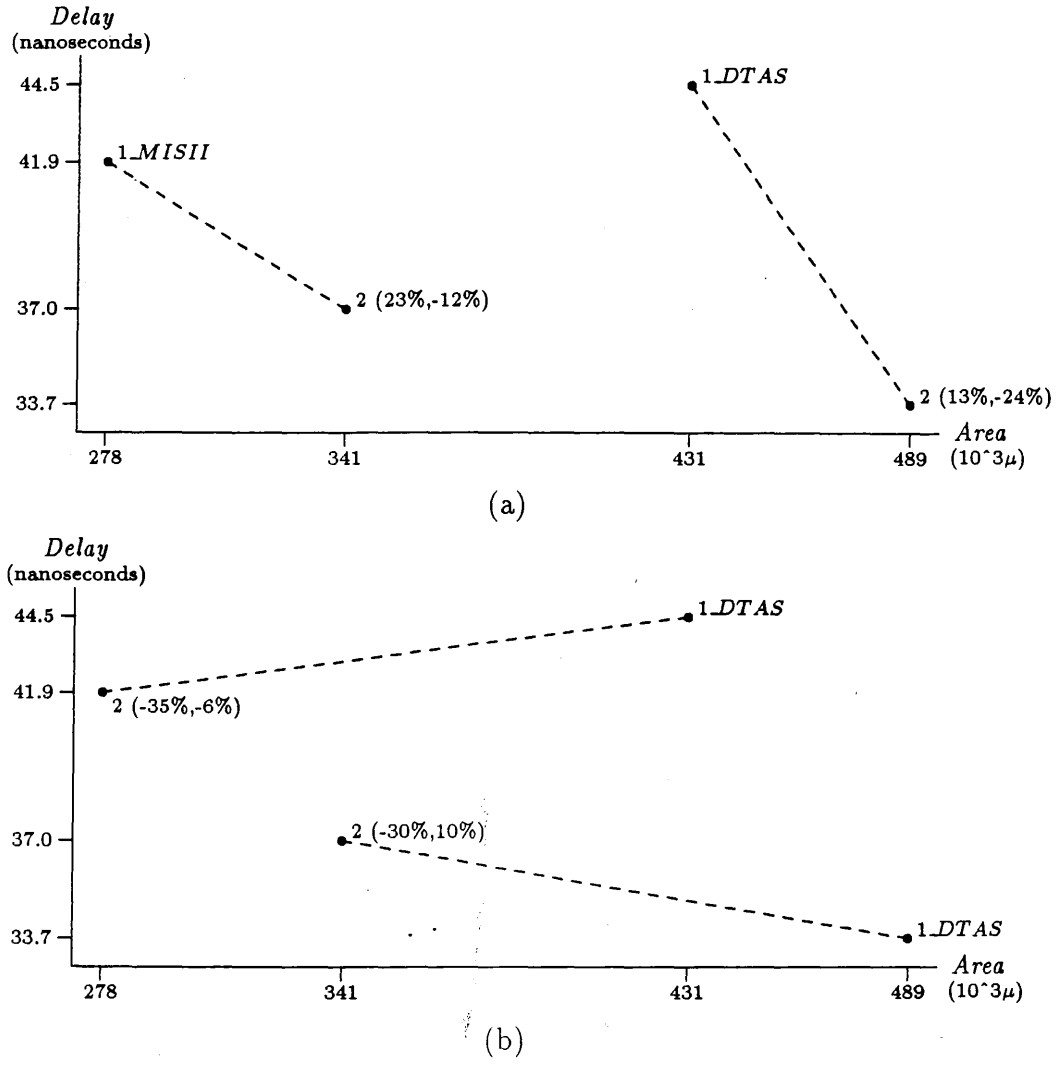


Figure 7.12: Experiment 3 – results for 8-bit/8-function ALU: (a) comparing DTAS designs to each other and MISII designs to each other; and (b) comparing DTAS designs to corresponding MISII designs.

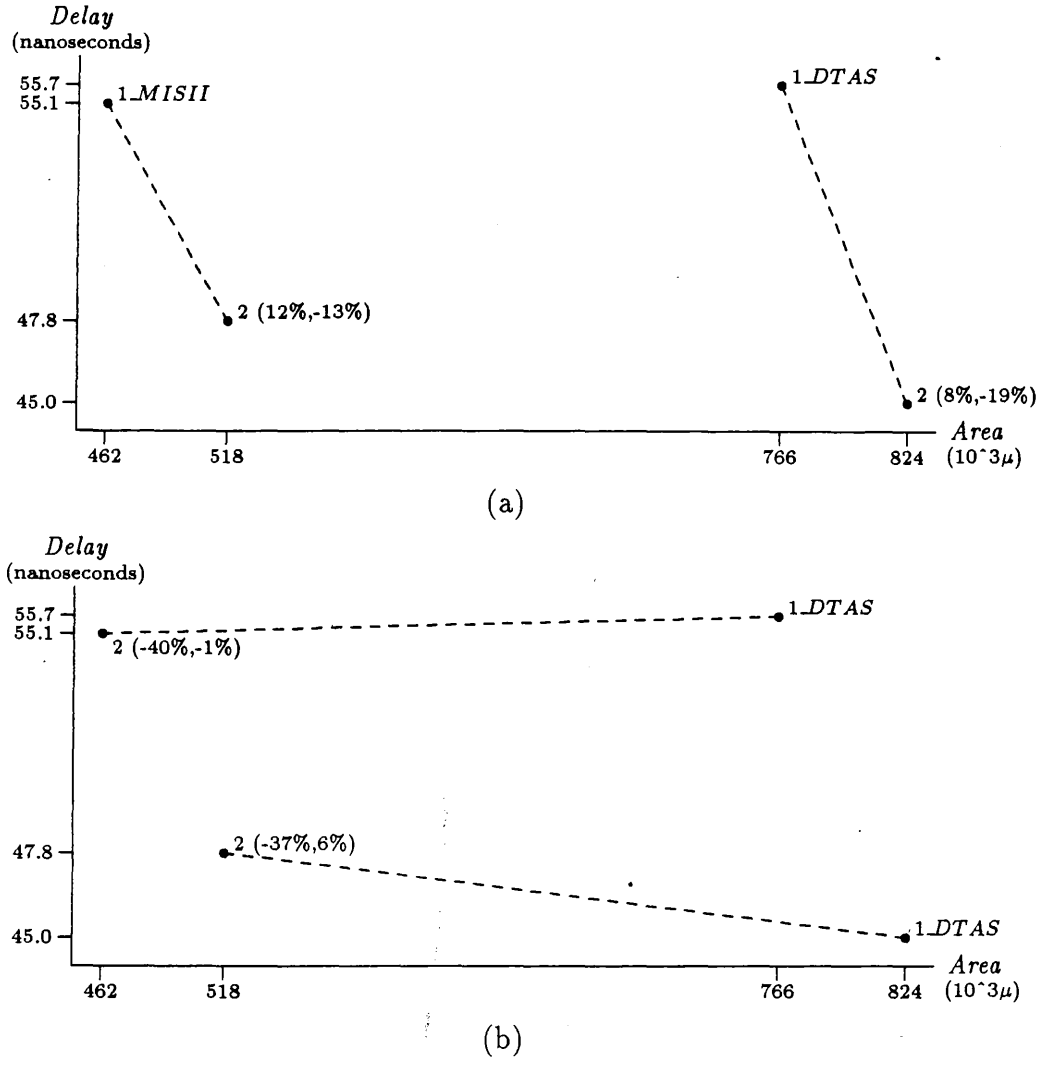


Figure 7.13: Experiment 3 – results for 8-bit/16-function ALU: (a) comparing DTAS designs to each other and MISII designs to each other; and (b) comparing DTAS designs to corresponding MISII designs.

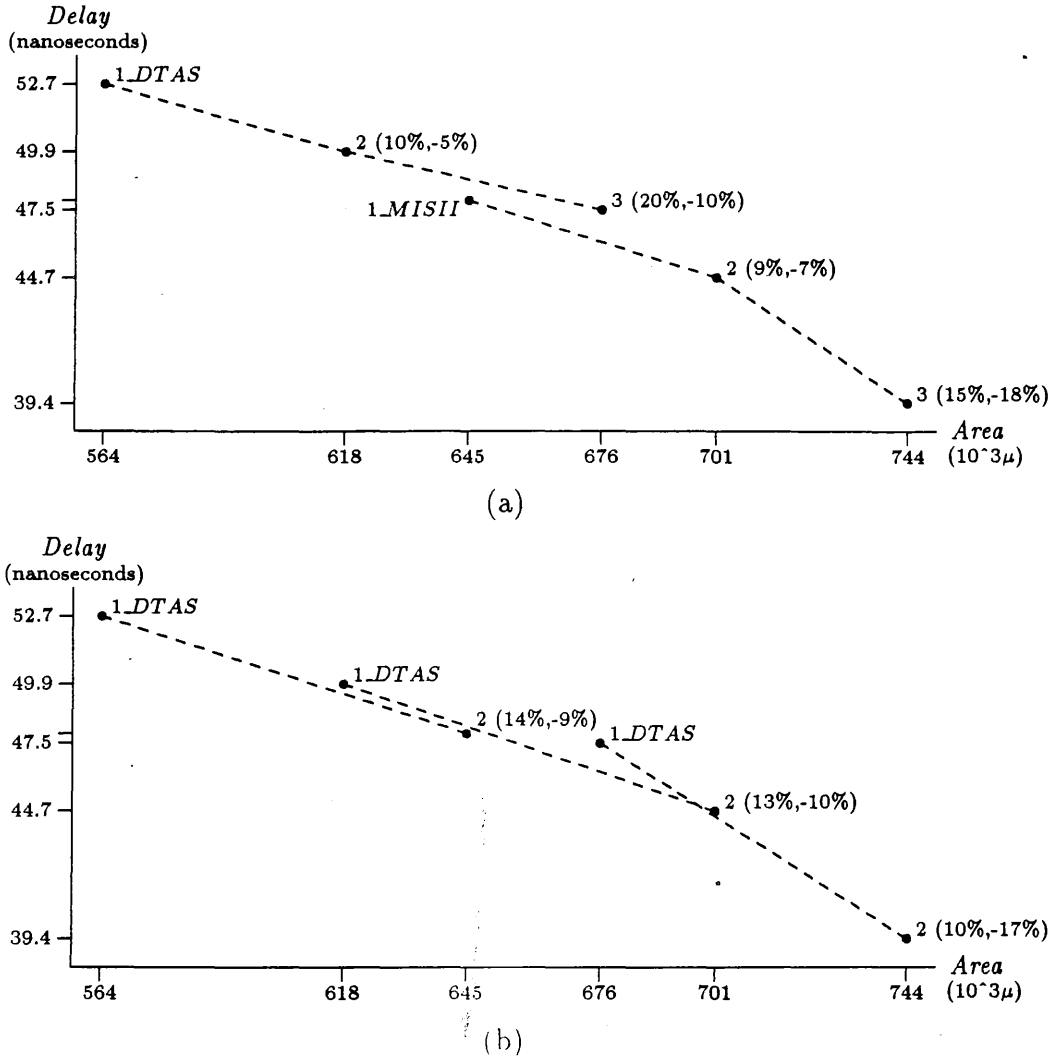


Figure 7.14: Experiment 3 – results for 8-bit multiplier: (a) comparing DTAS designs to each other and MISII designs to each other; and (b) comparing DTAS designs to corresponding MISII designs.

Example 3. Design results for the 8-bit multiplier are shown in Figure 7.14. Design 1 was a full matrix multiplier, design 2 was a tree of matrix multipliers with a ripple-carry adder, and design 3 was a tree of matrix multiplier with carry look-ahead adder.

As depicted in Figure 7.14(a), in both cases design 1 had the least area, design 3 had the shortest maximum delay, and design 2 was intermediate in area and delay, roughly by the same percentage. As depicted in Figure 7.14(b), MISII consistently optimized the delay of each of DTAS's designs, but at a cost of area. As one would expect from its performance on the adder example from above, MISII actually maked design 2 (the tree multiplier with a ripple-carry adder) a less desirable alternative to design 3.

7.3.4 Designing with Functional Cells

One of the hallmarks of the component decomposition algorithm is its ability to map designs into complex functional library cells. The component decomposition algorithm is unique in this regard. Complex functional cells are provided by ASIC vendors as optimized alternatives to commonly used logic components. The use of functional cells means that the component decomposition algorithm can generate higher-quality designs than if it uses functionally-equivalent configurations of simple Boolean cells.

In this set of experiments, I compare the quality of designs generated by the component decomposition algorithm with and without the use of complex functional cells. These experiments include the following examples:

1. 8-, 16-, 32-, and 64-bit adders;
2. 4-, 8-, 16-, 32-, and 48-bit ALUs that compute 8 and 16 functions;
3. 4-, 8-, and 16-bit multipliers.

Designs were mapped into two alternative cell libraries: (1) the subset of LSI Logic's macrocell library that only contains simple Boolean cells; and (2) the subset of LSI Logic's macrocell library containing both simple cells and complex functional cells, such as 4-bit adders, 4-bit CLAs, and 8-to-4 multiplexers. Results for each size component are shown on the same delay versus area graph; alternative designs for the same input specification are connected by dashed lines.

Example 1. Design results for the 8-, 16-, 32-, and 64-bit adders are shown in Figure 7.15. Designs depicted in Figure 7.15(a) were generated using Boolean cells only. Designs depicted in Figure 7.15(b) were generated using complex functional

cells. As can be seen, designs generated with complex functional cells were faster and smaller than designs generated with Boolean cells only.

With a few exceptions, DTAS typically generated two designs for the example adders. The smallest design (1) was a ripple-carry adder. In Figure 7.15(a), NAND-implemented adders were used; in Figure 7.15(b), the 1-bit adder cell FA1A was used. In Figure 7.15(a), the fastest design was a full carry look-ahead adder; however, for the 8- and 32-bit adder there were also intermediate designs that rippled two 4- and 16-bit carry look-ahead adders, respectively.

In Figure 7.15(b), the fastest designs for the 8-, 16-, and 32-bit adder were generated with the 4-bit adder cell FA4 and carry look-ahead generators CLA1 and CLA2. Because FA4's (with CLA1 or CLA2) are essentially 4-bit carry look-ahead adders that must be rippled, there comes a point at which a series of FA4's will have a longer maximum delay than a full carry look-ahead adder implemented with Boolean gates. This point first becomes apparent with the 32-bit adder, for which design 2 was identical to design 2 of Figure 7.15(a). For the 64-bit adder, DTAS did not generate a design using an FA4; instead, design 2 was the same carry look-ahead design appearing in Figure 7.15(a).

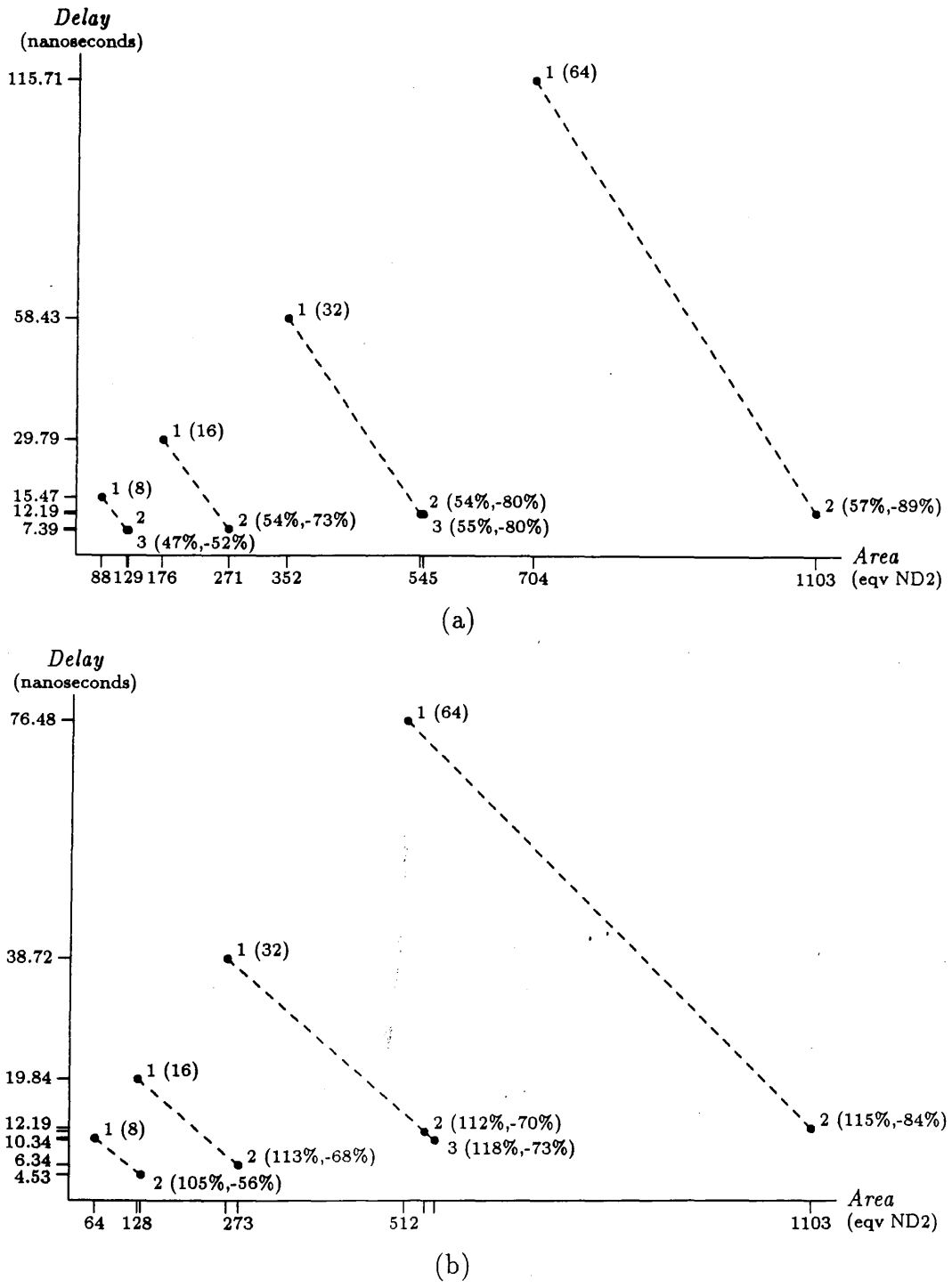


Figure 7.15: Experiment 4 – results for 8-, 16-, 32-, and 64-bit adders: (a) using boolean cells only; and (b) using complex functional cells.

Example 2. Design results for the 4-, 8-, 16, 32-, and 48-bit/8-function ALUs are shown in Figure 7.16; the functions computed include four arithmetic operations: addition, subtraction, increment, and decrement, and four logic operations: AND, OR, NAND, and NOR. Designs depicted in Figure 7.16(a) were generated using Boolean cells only; those in Figure 7.16(b) were generated using functional cells.

The designs depicted in Figure 7.16(a) look distinctly different than the designs generated for the same ALUs using the MCNC cell library (Figure 7.8). For the LSI library, Boolean implementations of the ALU using the segregated style never outperformed designs using the integrated style; for the MCNC library, designs using the segregated style were always the fastest by a small percent. Design alternatives appearing in Figure 7.16(a) resulted from the differences in adder styles. For instance, of the four designs for the 32-bit ALU, design 1 used a ripple-carry adder, design 2 used an adder that ripples eight 4-bit carry look-ahead adders, design 3 used an adder that ripples four 8-bit carry look-ahead adders, and design 4 used an adder that ripples two 16-bit carry look-ahead adders.

When DTAS was provided with functional library cells, it typically generated three alternative designs for each ALU, as depicted in Figure 7.16(b). For each case, design 1 used an integrated style and a ripple-carry adder, design 2 used an integrated style and a Boolean carry look-ahead adder (these design also appears in Figure 7.16(a)); each design 3 used a segregated style. The designs in Figure 7.16(b) include ALUs using the segregated style because DTAS was able to take advantage of the 4-bit library adder FA4. In generating designs using the integrated style, DTAS used the 1-bit library adder FA1A in its smallest designs, producing designs that were significantly smaller and faster than those appearing in Figure 7.16(a). DTAS could not use an FA4 in an integrated ALU because the integrated style requires an adder with a carry enable input.

Results for the 16-function ALUs are shown in Figure 7.17. Again, the functions computed include four arithmetic operations: addition, subtraction, increment, and decrement, four comparison operations: equal, less than, greater than, and equals zero, and eight logic operations: AND, OR, NAND, NOR, XOR, XNOR, INV (left-most input), and IMPLICATION (leftmost input).

As depicted in Figure 7.17(a), the addition logic operations makes the integrated ALU sufficiently complex that the fastest design was always computed using the segregated ALU style. However, this style was still only a small percentage faster than the integrated style with a carry look-ahead adder. Otherwise, these results are similar to those depicted in Figure 7.16. Again, as depicted in Figure 7.17(b), using complex functional cells DTAS was able to generate faster and smaller designs than when using Boolean cells only.

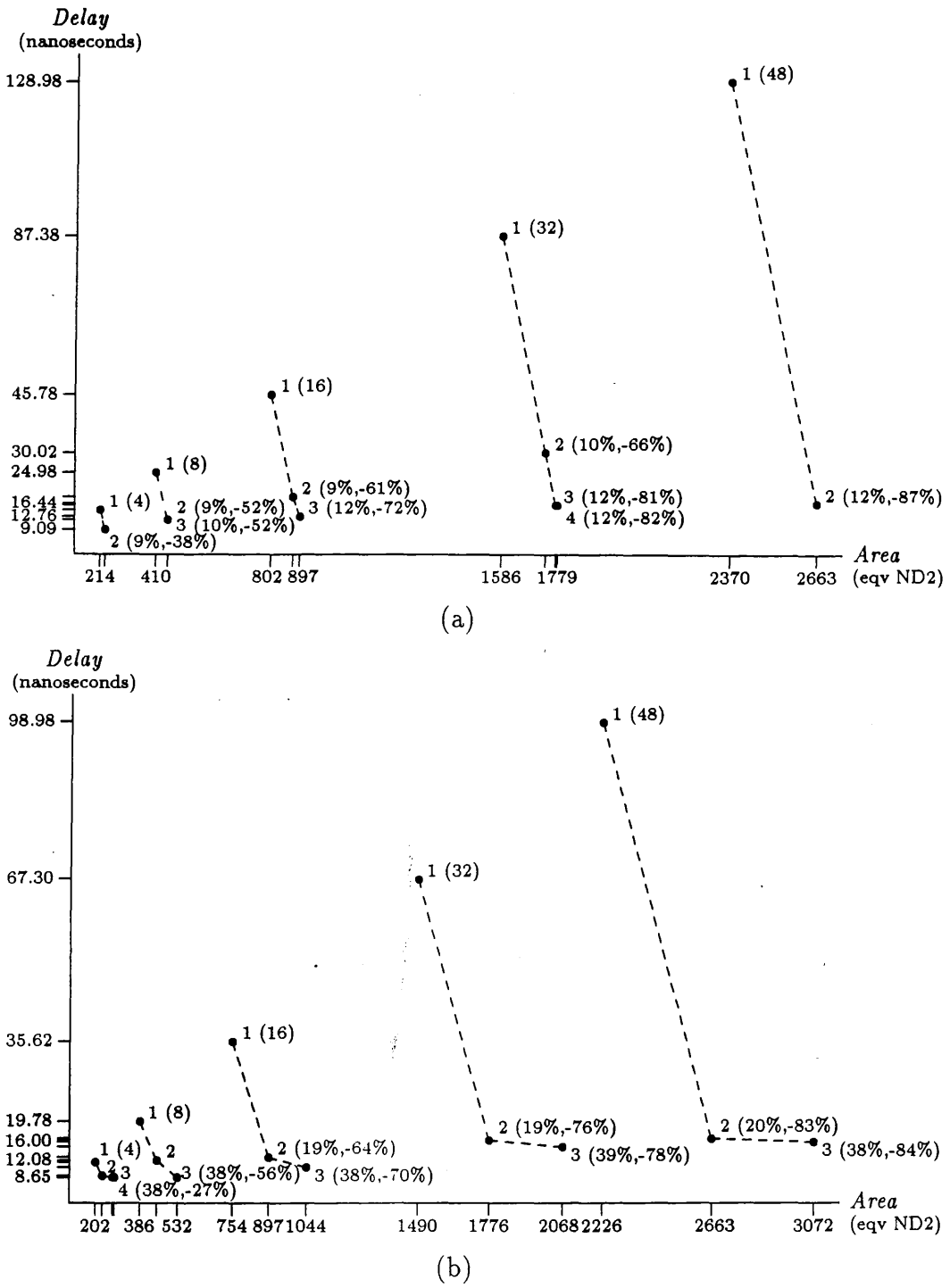


Figure 7.16: Experiment 4 – results for 4-, 8-, 16-, 32-, and 48-bit/8-function ALUs: (a) using boolean cells only; and (b) using complex functional cells.

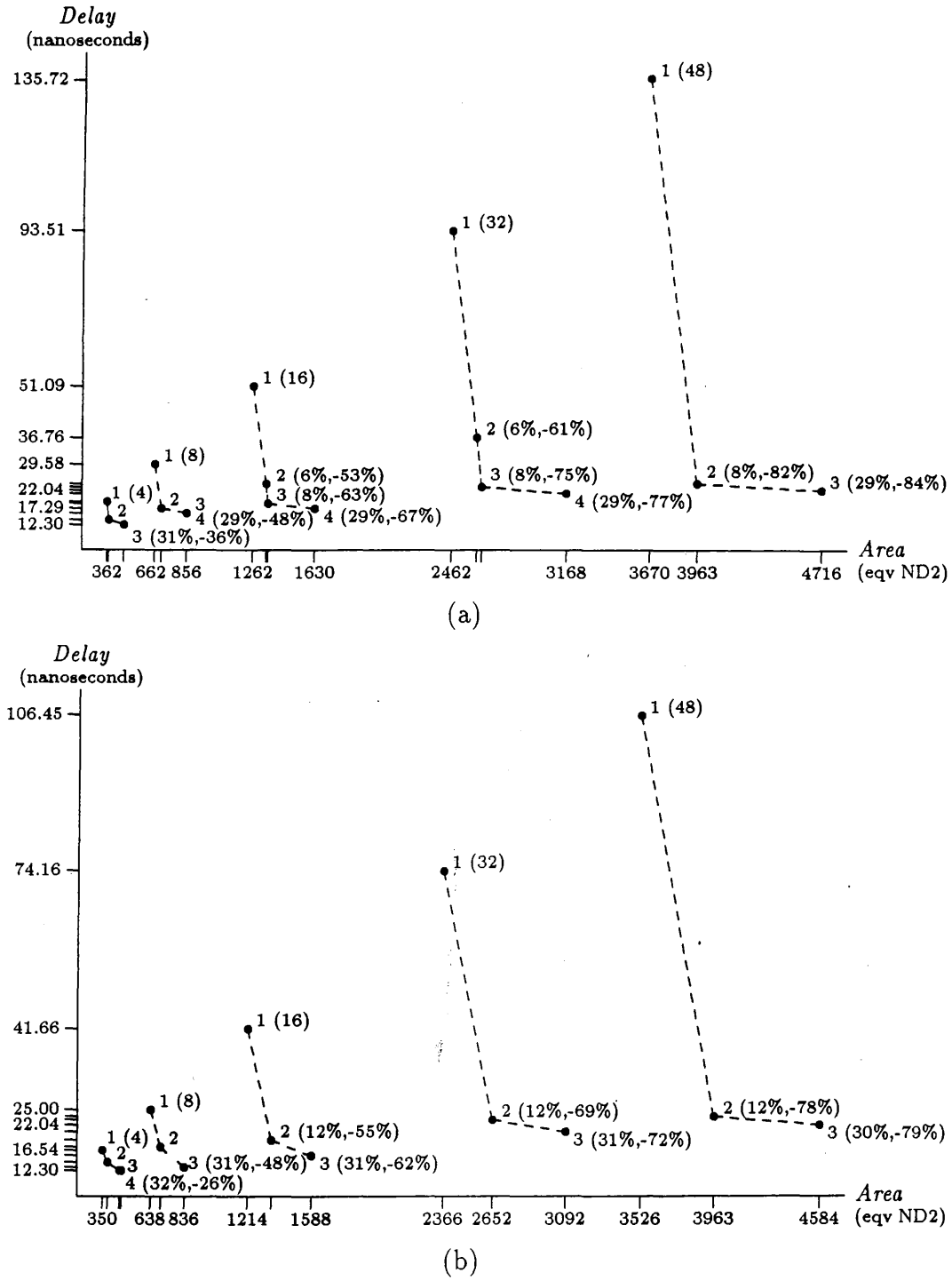


Figure 7.17: Experiment 4 – results for 4-, 8-, 16-, 32-, and 48-bit/16-function ALUs: (a) using boolean cells only; and (b) using complex functional cells.

Example 3. Design results for the 4-, 8-, and 16-bit multipliers are shown in Figure 7.18. As seen earlier in Figure 7.10, a wide range of designs were generated for each multiplier specification. In each case, design 1 was always a full matrix multiplier, and the last design was always a full tree multiplier with carry look-ahead adders in each tree. Design 2 was typically a tree of four matrix multipliers and a ripple-carry adder, design 3 was typically a tree of matrix multiplies and a carry look-ahead adder. Other designs use a tree of tree of matrix style with carry look-ahead adders at the highest level.

The designs depicted in Figure 7.18(a) were generated using Boolean cells only; those depicted in Figure 7.18(b) were generated using functional cells. In both cases, the matrix multiplier was always the same Boolean implementation. The difference between the use of Boolean cells and functional cells only becomes apparent with tree multipliers. Using complex function cells allows DTAS to generate a wider range of hybrid Wallace-tree multipliers that were smaller and significantly faster than can be generated with Boolean cells alone.

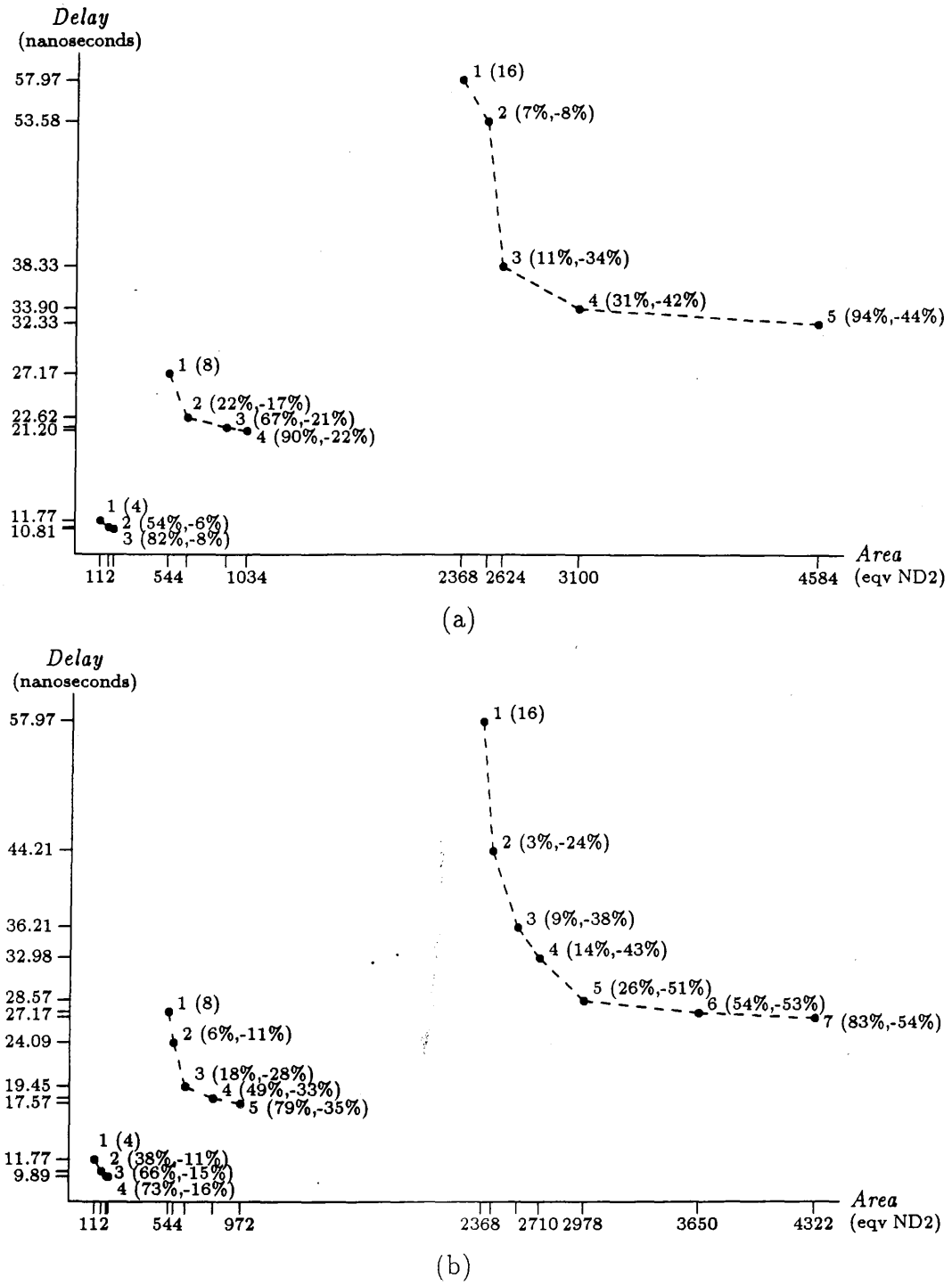


Figure 7.18: Experiment 4 – results for 4-, 8-, and 16-bit multipliers: (a) using boolean cells only; and (b) using complex functional cells.

7.4 Summary

I have claimed that the component decomposition algorithm has the following significant benefits as an approach to component generation:

1. It supports the use of complex functional library cells;
2. It effectively compares design alternatives.

I have validated these claims with empirical results from four sets of experiments.

- The first set of experiments showed how the search control principles of the component decomposition algorithm allowed DTAS to find desirable designs from design spaces that were computationally intractable to enumerate.
- The second set of experiments showed how the encapsulation of design styles in decomposition methods allowed DTAS to compare design alternatives and find ranges of designs that make desirable trade-offs between area and delay.
- The third set of experiments showed how the design generated by DTAS compared to the same designs when passed through the MISII logic optimizer. DTAS's designs were often close in performance to designs generated with MISII.
- The fourth set of experiments showed how the use of functional decomposition and functional specification allowed DTAS to map designs into libraries of complex functional cells, generating higher-performance designs than was possible with Boolean cells alone.

Results were collected using the DTAS component generation system, and designs were mapped into either an MCNC benchmark standard cell library or a macrocell library from LSI Logic, Inc. The results presented in this chapter support my claims concerning the effectiveness of the component decomposition algorithm.

Chapter 8
Technology Compilation and LOLA

In this chapter, I present a formal definition of the technology compilation algorithm and its implementation. First, I give a conceptual overview of the algorithm; then, I define the algorithm and its data structures; finally, I describe its implementation in the LOLA technology adaptation system. Examples are provided to clarify concepts as needed. Detailed demonstrations are provided in Chapter 9.

8.1 Overview

The component decomposition algorithm, as defined in Chapter 4, is technology independent. As implemented in DTAS, it depends (1) on a set of tool-specific decomposition methods for mapping the functional units used by a high-level synthesis tool into its generic component models, and (2) on a set of library-specific decomposition methods (also known as *construction methods*) for mapping generic classes of components into particular ASIC cell library. The technology compilation algorithm demonstrates an approach to automatically generating construction methods for commonly occuring classes of library cells.

The basic operation of the technology compilation algorithm is to examine the cells available in a given ASIC library and, for cells that it recognizes, to generate methods that are specific to those cells. Knowledge of how to generate construction methods for particular classes of cells is based on fundamental principles of logic design, as described in Chapter 3, Section 3.7.

Design principles and their use in generating methods are represented with *acquisition templates*. The applicability of an acquisition template is described in terms of classes of cells that *are* and *are not* available in a given cell library. In particular, acquisition templates pair two sets of component specification patterns with a procedure for defining decomposition methods. One set of specification patterns is

called the *include* set; the other is called the *exclude* set. An acquisition template is applicable to any distinct set of library cells whose specifications are described by the include set, given that there is not another set of library cells whose specifications are described by the exclude set.

When an applicable acquisition template is *expanded*, it generates one or more decomposition methods, where each method pairs a component specification pattern to a procedure for configuring a netlist of connected subcomponents. Typically, the generate methods will mark particular subcomponents as instances of the library cells matched in the include set, thus making the method library-specific, in which case it can be referred to as a construction method. The technology compilation algorithm iterates over the defined acquisition templates. For each template, the algorithm finds every distinct set of cells that match the include set of the template without another set of cells matching the exclude set; the template is expanded for each such set of cells.

To illustrate these concepts further, consider a cell library that contains a 2-bit adder cell (ADD2), a 4-bit adder cells (ADD4), and and a 16-bit adder cell (ADD16). The component decomposition algorithm will already map 2-bit, 4-bit, and 16-bit adders into the available library cells. Construction methods are needed that map n-bit adders into appropriate configurations of library adders, including

1. A method that maps n-bit adders ($n > 16$) into $\frac{n}{16}$ ADD16's;
2. A method that maps n-bit adders ($16 > n > 4$) into $\frac{n}{4}$ ADD4's;
3. A method that maps 3-bit adders into an ADD2 and a 1-bit adder.

The first construction method can be generated by an acquisition template whose include set matches any p-bit adder cell and whose exclude set matches any q-bit adder cell, such that $q > p$. This template would only be applicable to the ADD16 cell, because there is no larger adder cell to match the exclude set. The method generated by this template would map n-bit adders into $\frac{n}{p}$ ADD16's plus some remainder.

The other two construction methods can be generated by an acquisition template whose include set matches any p-bit adder cell and any q-bit adder cell, such that $q > p$, and whose exclude set matches any r-bit adder cell, such that $q > r > p$. This template is applicable to two distinct sets of cells: ADD2 and ADD4, and ADD4 and ADD16. Although the set of cells ADD2 and ADD16 also matches the include set of this template, the template is not applicable because ADD4 matches the exclude set. The method generated by this template would map n-bit adders, where $q > n > p$, into $\frac{n}{p}$ p-bit adder cells plus some remainder. Instantiating p to 4 and q to 16 gives the second desired construction method; instantiating p to 2 and q to 4 gives the third.

As defined by the technology compilation algorithm, the LOLA technology adaptation performs an exhaustive search, expanding all applicable acquisition templates. There are plans to include two addition phases to LOLA: one that evaluates the performance of DTAS given the generated methods; and another that constrains methods that do not lead to desirable designs. However, these two phases have not been implemented.

Because of the great degree to which functional units can be customized, it is not possible to anticipate all possibly occuring library cells. Thus, the approach to technology adaptation described is only a partial solution. It can, however, be used to generate library-specific decomposition methods for prototypical cells, which should account for many of the cells in a library. It can also be used to detect atypical cells for which no decomposition methods were generated and for which library-specific methods must be generated by other methods.

8.2 The Technology Compilation Algorithm

The technology compilation algorithm augments the component decomposition algorithm and reuses all of its existing data structures, expressions, and conditions, which were defined earlier in Chapter 4, as well as several of its actions. The technology compilation algorithm adds one new data structure and one new action, the textual formats of which are outline in Figure 8.1. Their semantics are defined below.

8.2.1 Acquisition Templates

The principle data structure of the technology compilation algorithm is the *acquisition template* (or *template*), the format of which is shown in Figure 8.1. An acquisition template describes the conditions under which one or more library-specific decomposition methods are to be acquired as well as how those methods are to be defined.

The conditions under which methods are to be acquired are described in terms of the cells that are and are not available in the target cell library. The includes (*incls*) and excludes (*excls*) fields of an acquisition template contain component specification patterns that are matched against the component specifications describing cells in the target library. The *test* field associated with each include and exclude entry is a condition on the values bound to variables of the component specification pattern.

If the specification patterns in the includes field of an acquisition template are matched by a distinct set of library cells and unless all of the specification patterns in

Structure	Representation
template	<*incls*, *excls*, *test*, *body*>
incls	[<$*var*, *cspec*, *test*>*]
excls	[<*cspec*, *test*>*]
test	[*cond*]
body	[*action**]

Actions

[*action**]
BIND($*var*, *value*)
CASE([<*cond*, *action*>*])
LOOP([*iterator**], *action*)
  iterator ::= { STEP($*var*, *init*, *limit*, *step*) | IN($*var*, *list*) }*
ACQUIRE-METHOD(*head*, *test*, *body*)

Figure 8.1: Structure of acquisition templates and actions.

the excludes field are matched by another distinct set of library cells (not matched by any entry in the includes field), then the body of the acquisition template is expanded, which results in the generation of one or more decomposition methods.

Each component specification pattern in the includes field of an acquisition template is paired with a variable. During the matching process, this variable is bound to the library-specific name of the matched cell. This variable can be used to identify the cell by name in the body of a generated method.

The component specification patterns of the includes and excludes fields of an acquisition template are similar to those found in the heads of decomposition methods, i.e., where variables can be used in place of port widths and attribute values. There are two important differences. First, variables in an aquisition template are denoted as symbols prefixed with a dollar sign (e.g., `$n`); they will be referred to as *dollar-sign* variables. The variables of decomposition methods, denoted as symbols prefixed with a question mark (e.g., `?n`), are treated as literals. Second, dollar-sign variables can be used to match the type field of a component specification or one of its ports or attributes; a dollar-sign variable delimited by a pair of dashes (e.g., `-$p-`) can be used to match a sequence of zero or more ports.

An acquisition template is expanded by executing the actions in its body. The actions of an aquisition template are outlined in Figure 8.1. These include the BIND, LOOP, and CASE actions of decomposition methods, which have similar semantics, plus a new ACQUIRE-METHOD action. The ACQUIRE-METHOD action takes as its arguments the head, test, and body of a decomposition method. The semantics of the ACQUIRE-METHOD action are (1) to replace instances of dollar-sign variables found in its head, test, and body arguments with the values to which these variables

are bound and (2) to create and return a decomposition method containing these instantiations as its head, test, and body.

To illustrate these concepts, consider Figure 8.2. A simple acquisition template is shown in Figure 8.2(a). This template is applicable when the cell library contains an n-input NAND gate and no n-input AND gate. When expanded, the body of this template generates a decomposition method that implements such an AND gate with the library NAND and a generic inverter (INV). Thus, if a cell library contains a 2-input NAND gate (ND2) shown in Figure 8.2(b) and no corresponding 2-input AND gate, then the acquisition template is applicable and can be expanded, resulting in the generation of the method shown in Figure 8.2(c).

```
<[ <$cell, <NAND,[<I0,$n>],[<O0,1>],[]> [] >],
 [ <<AND,[<I0,$n>],[<O0,1>],[]> [] >],
 [ ],
 [ ACQUIRE-METHOD( <AND,[<I0,$n>],[<O0,1>],[]> , [ ],
    [
       CONNECT-SRC(A0,NETLIST,I0,0),
       CONNECT-SRC(A1,NETLIST,I0,1),
       CONNECT-SNK(Z,NETLIST,O0,0),
       ADD-CELL($cell,  <NAND,[<I0,$n>],[<O0,1>],[]>),
       ADD-CSPEC(INV,   <INV,[<I0,1>],[<O0,1>],[]> ),
       ADD-MODULE(NAND_0, $cell),
       ADD-MODULE(INV_0, INV),
       CONNECT-SNK(A0,NAND_0,I0,0),
       CONNECT-SNK(A1,NAND_0,I0,1),
       CONNECT-SRC(B,NAND_0,O0,0),
       CONNECT-SNK(B,INV_0,I0,0),
       CONNECT-SRC(Z,INV_0,O0,0)
    ])
]>
```

(a)

```
<ND2, <NAND,[<I0,2>],[<O0,1>],[ ]>, [<AREA,...>,<DELAY...>,<COST,...>]>
```

(b)

```
<<AND,[<I0,2>],[<O0,1>],[]>  ,
 [ ],
 [
  CONNECT-SRC(A0,NETLIST,I0,0),
  CONNECT-SRC(A1,NETLIST,I0,1),
  CONNECT-SNK(Z,NETLIST,O0,0),
  ADD-CELL(ND2,   <NAND,[<I0,2>],[<O0,1>],[]> ),
  ADD-CSPEC(INV,   <INV,[<I0,1>],[<O0,1>],[]> ),
  ADD-MODULE(NAND_0, $cell),
  ADD-MODULE(INV_0, INV),
  CONNECT-SNK(A0,NAND_0,I0,0),
  CONNECT-SNK(A1,NAND_0,I0,1),
  CONNECT-SRC(B,NAND_0,O0,0),
  CONNECT-SNK(B,INV_0,I0,0),
  CONNECT-SRC(Z,INV_0,O0,0)
 ]>
```

(c)

Figure 8.2: Implementing n-input AND gate from n-input NAND cell: (a) sample acquisition template; (b) library cell specification for 2-input NAND (ND2); and (c) acquired decomposition method.

```
ACQUIRE-METHODS( )
  methods = [ ];
  for ∀ template ∈ TEMPLATE-LIBRARY do
    ADD GENERATE-METHODS(template) to methods;
  endfor
  RETURN(methods);

GENERATE-METHODS(template)
  methods = [ ];
  for ∀ ⟨cells,bdgs⟩ ∈ LIBRARY-INCLUDES(template.incls, [ ], [ ]) do
    if template.excls is not empty then
      if LIBRARY-EXCLUDES(template.excls, cells, bdgs) then
        ADD EXPAND-TEMPLATE(template, bdgs) to methods;
      endif
    endif
  endfor
  RETURN(methods);
```

Figure 8.3: Definition of ACQUIRE-METHODS and GENERATE-METHODS.

8.2.2 The Algorithm

The technology compilation algorithm generates a set of library-specific decomposition methods by expanding all instantiations of the given acquisition templates against the target cell library. I factor my definition of the technology compilation algorithm between five functions: ACQUIRE-METHODS, GENERATE-METHODS, LIBRARY-INCLUDES, LIBRARY-EXCLUDES, and MATCH-CELL. In their definition, lists are delimited with square brackets and tuples with angle brackets; an at-sign (@) denotes a list append operation, e.g., `[a b]@[c]` returns `[a b c]`.

The top-most function, ACQUIRE-METHODS, is defined in Figure 8.3. This function returns a list of library-specific methods generated from each of the acquisition template (*template*) defined in TEMPLATE-LIBRARY. Methods are generated from a template with the function GENERATE-METHODS. Once returned by ACQUIRE-METHODS, methods can be input to other processes for evaluation, or they can be output directly to a file.

The function GENERATE-METHODS is also defined in Figure 8.3. It operates by first using LIBRARY-INCLUDES to find all combinations of library cells that match the component specification patterns in the includes field of the input acquisition template (*template*). Each combination (*cells*) is accompanied by the binding list (*bdgs*) generated in unifying the dollar-sign variables of the specification patterns to the component specifications of the matched cells.

```
LIBRARY-INCLUDES(incls, seen, bdgs)
  if incls is empty then RETURN(⟨seen,bdgs⟩); endif
  match-list = [ ];
  first = incls.first;
  for ∀ cell ∈ CELL-LIBRARY do
    if cell is not member of seen then
      new-bdgs = MATCH-CELL(cell.cspec, first.cspec, first.test, bdgs);
      if MATCH-CELL succeeds then
        ADD ⟨first.var, cell.name⟩ to new-bdgs;
        ADD LIBRARY-INCLUDES(incls.rest, [cell] seen, new-bdgs) to match-list;
      endif
  endfor
  RETURN(match-list);
```

Figure 8.4: Definition of LIBRARY-INCLUDES.

For each combination of cells and their bindings, GENERATE-METHODS calls LIBRARY-EXCLUDES, which tests if the excludes field of the acquisition template can be instantiated by other library cells not already matched by the includes field. If not, then the template is applicable and can be expanded with EXPAND-TEMPLATE, which executes the actions in the body of the template and returns all methods instantiated with the ACQUIRE-METHOD action.

The function LIBRARY-INCLUDES, defined in Figure 8.4, recursively compares the component specification patterns in the includes set (*incls*) to the cells in the target cell library (CELL-LIBRARY). Each element of *incls* is a triple

$$<\$var, cspec, test>,$$

where $\$var$ is a dollar-sign variable, *cspec* is a component specification pattern, and *test* is a condition. The input *seen* is a list of library cells that have already been matched by specification patterns appearing earlier in the includes set; and *bdgs* is a binding list for dollar-sign variables unified in these earlier matches.

LIBRARY-INCLUDES returns a list of pairs **<*seen*, *bdgs*>**, where each *seen* is a list of library cells that matched the elements in *incls* and *bdgs* is a binding list for the unifications made for that match. If *incls* is an empty list, then all previous elements of the include set have been successfully matched by the cells in *seen* with binding list ***bdgs***, so the pair of *seen* and *bdgs* is returned as a single-element list. Otherwise, it is necessary to find all library cells, not already in *seen*, that match the component specification pattern at the head of *incls* and combine these with all matches of the entries in the tail of *incls*. For each matching cell, the dollar-sign variable paired to the specification pattern is bound to the cell's library-specific name. Finally, LIBRARY-INCLUDES returns the accumulated instantiations.

```
LIBRARY-EXCLUDES(excls, seen, bdgs)
  if excls is empty then FAIL; endif
  first = excls.first;
  for ∀ cell ∈ CELL-LIBRARY do
    if cell is not member of seen then
      new-bdgs = MATCH-CELL(cell.cspec, first.cspec, first.test, bdgs);
      if MATCH-CELL succeeds then
        if LIBRARY-EXCLUDES(excls.rest, [cell] seen, new-bdgs) then FAIL; endif
      endif
  endfor
  SUCCEED;
```

Figure 8.5: Definition of LIBRARY-EXCLUDES.

```
MATCH-CELL(cspec, cptrn, test, bdgs)
  bdgs = UNIFY_$V(cspec.type, cptrn.type);
  if UNIFY_$V fails then FAIL; endif
  bdgs = UNIFY-PORTS_$V(cspec.iports, cptrn.iports, bdgs);
  if UNIFY-PORTS_$V fails then FAIL; endif
  bdgs = UNIFY-PORTS_$V(cspec.oports, cptrn.oports, bdgs);
  if UNIFY-PORTS_$V fails then FAIL; endif
  bdgs = UNIFY-ATTRS_$V(cspec.attrs, cptrn.attrs, bdgs);
  if UNIFY-ATTRS_$V fails then FAIL; endif
  if EVAL_$V(test, bdgs) fails then FAIL; endif
  SUCCESS: RETURN(bdgs);
```

Figure 8.6: Definition of MATCH-CELL.

The function LIBRARY-EXCLUDES, defined in Figure 8.5, is similar to the function LIBRARY-INCLUDES, with the exception that failure indicates that all specification patterns in the excludes set (*excls*) are matched by distinct set of library cells. It recursively compares each component specification pattern in *excls* to the library cells not already in *seen* and succeeds if, for any specification, there is no matching set of library cells.

Finally, the function MATCH-CELL, shown in Figure 8.6, defines the top level of the unification process for matching a component specification pattern (*cptrn*) to the component specification of a library cell (*cspec*). A match is successful if the types of the two specifications unify, if the input and output ports unify, if the attributes unify, and if *test* evaluates successfully. The functions defining the unification of types and ports are not defined here; they are similar to the unification functions defined in Chapter 4, with the exception that dollar-sign variables can match sequences of ports.

8.3 The LOLA Technology Adaptation System

I have implemented the technology compilation algorithm in the Logic Learning Assistant (LOLA). LOLA operates as a preprocess to component generation with DTAS. Its inputs include the LIB file specification of the target cell library and a set of acquisition templates. The output consists of library-specific decomposition methods for implementing generic components with library cells. This process is performed in advance of synthesis.

8.3.1 The LOLA System Architecture

The fundamental system architecture of LOLA and its relationship to the DTAS component generation system is shown in Figure 8.7. The input to DTAS is a high-level component specification and the output is a set of physical designs using cells from a given ASIC library. Designs are generated as a hierarchical netlist using generic decomposition methods and mapped into library cells using decomposition methods that are library specific. The purpose of LOLA is to ensure that DTAS has access to decomposition methods that are appropriately biased towards the available library cells. This interaction allows DTAS to take full advantage of the cells in the given ASIC library, with the potential of improving its design quality.

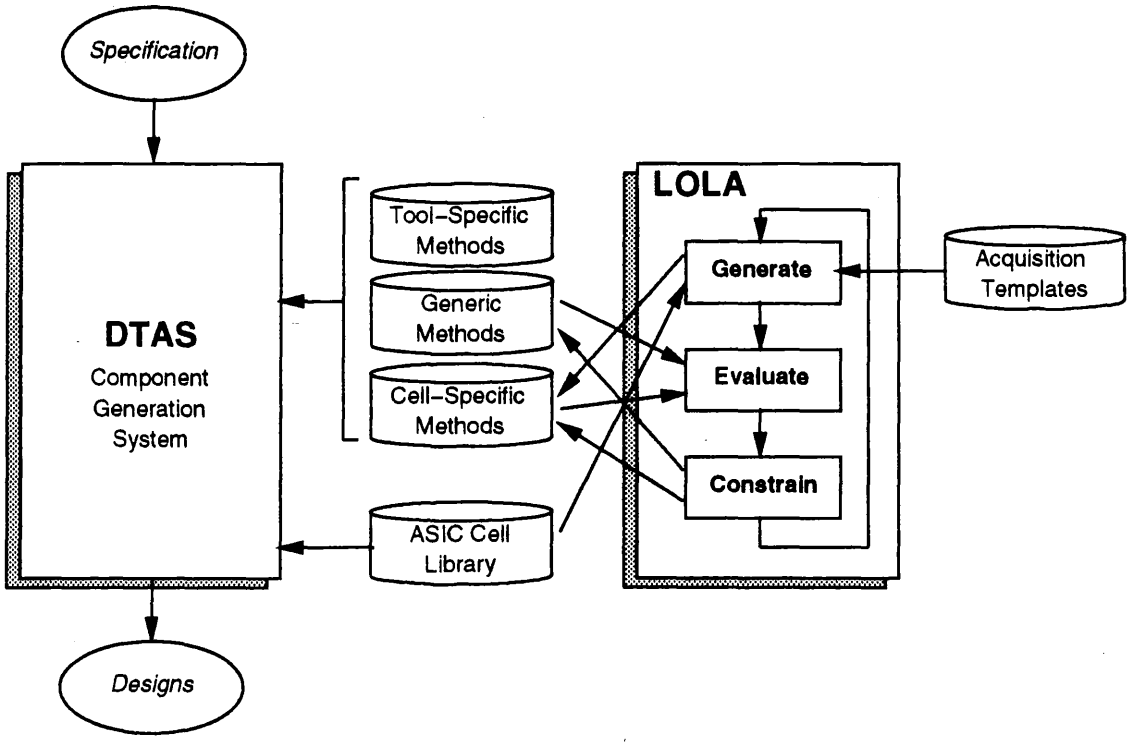


Figure 8.7: Relationship of LOLA to DTAS.

The decomposition methods generated by LOLA essentially generalize the fixed features of library cells to match to the variable features of classes of generic components. The fixed features of a cell correspond to the type and width of its inputs and outputs and the functions it performs. For example, a library might contain a 4-bit adder cell, the fixed features of which include two 4-bit data inputs, one 4-bit data output, a carry input, and a carry output, as well as the fact that it performs the function ADD. This cell can be used in a decomposition method that implements a generic adder with two n-bit data inputs, one n-bit data output, and optional carry input and output; it can also be used in a decomposition method that implements a n-bit subtractor.

After generating new methods, LOLA is intended to evaluate the entire set of methods, both library specific and generic. Evaluation is used to identify the conditions under which existing or newly generated methods fail to generate desirable designs in terms of area or delay. These conditions become constrains on the unproductive methods, limiting their applicability, allowing DTAS to avoid otherwise expanding large portions of the design space. (The evaluation and constraint components of LOLA have not been implemented at this time.)

```
if library  [ includes { var : cspec [ where cond ] }+ ]
  [ and ] [ excludes { cspec [ where cond ] }+ ]
  => action

action ::= [ action + ]
       ::= let { var := expr }+
       ::= if cond action [ else action ]
       ::= for iter { as iter }* action
       ::= acquire { method + }
```

Figure 8.8: Syntactic form of acquisition templates.

8.3.2 Acquisition Templates

The implementation of LOLA augments the DTAS design language with a syntax for acquisition templates. The general form of this syntax is defined in Figure 8.8. Using this syntax, the example acquisition template seen earlier in Figure 8.2(a) can be written as shown in Figure 8.9(a). When the `acquire` action is executed, LOLA generates a decomposition method from its argument after replacing all dollar-sign variables with the value to which they are bound.

For instance, when the cell library contains the 2-input NAND gate ND2, shown in Figure 8.9(b), and no 2-input AND gate, then the acquisition template in Figure 8.9(a) can be expanded, with `$n` bound to 2. All instances of `$n` in the argument of the `acquire` action are replaced by its bound value, generating the decomposition method shown in Figure 8.9(c). Decomposition methods variables, introduced by a question mark, e.g., `?n`, are treated as literals within the body of the acquisition template and are not bound nor evaluated by LOLA.

8.3.3 The Acquisition Process

LOLA is intended to acquire new decomposition methods through a process of generate, evaluate, and constrain. As new methods are generated, LOLA runs a series of tests to evaluate the quality of designs generated with these methods as opposed to designs generated by existing methods. For each case in which a new or existing method does not generate a design that outperformed another method (e.g., in terms for area or delay), the case is generalized and appended as a test of the method, which ensures that the method will not be fired under conditions when it is known to generate inferior designs.

For example, consider the example seen above where an n-bit NAND gate can be constructed from a 2-bit library NAND. If the library also contained a 4-bit NAND that is smaller and faster than the configuration of 2-bit NAND gates generated by the acquired decomposition method, then LOLA will add the constraint that `?n` not be bound to 4.

The generate-evaluate-constrain cycle is intended to eliminate many overly general methods that would otherwise force DTAS to do unnecessary expansion and mapping. Decomposition methods that generate designs using complex library cells will typically subsume methods that decompose into functionally equivalent configurations of simpler cells. In most cases, this means that the design space will be successively shallower with the introduction of complex library cells. As DTAS is supplied libraries of increasingly complex library cells, it should actually generate fewer alternative designs; it should generate designs of higher quality; and it should generate them faster.

```
If library includes
    $cell: NAND(I0^$n)
                (O0)
and excludes
    AND (I0^$n)
        (O0)
=>
    acquire
      {
        AND (A^$n)
            (Z)
          ->
              $cell:  NAND(A[0..$n])
                           (B)
                      INV(B)
                          (Z)
      }
```

(a)

```
ND2:  NAND(A B)
             (Z)
       :load ...
       :area ...
       :delay ...
```

(b)

```
AND (A^2)
    (Z)
  ->
     ND2:  NAND(A[0..1])
                (B)
           INV(B)
               (Z)
```

(c)

Figure 8.9: Example of LOLA syntax: (a) sample acquisition template; (b) 2-input NAND library cell (ND2); and (c) resulting decomposition method.

Chapter 9
Validating Technology Compilation

In this chapter, I demonstrate the utility of the technology compilation algorithm, as implemented in the LOLA technology adaptation system. First, I present an overview of the demonstrations; then, I step through each of four demonstration sets; finally, I present a summary of the demonstration results. These demonstrations are intended to informally validate the effectiveness of the technology compilation algorithm. In particular, they show how LOLA can be used to maintain the proficiency, completeness, and coverage of the DTAS component generation system as its target ASIC cell library changes over time.

9.1 Demonstrations

In this dissertation, I claim that a symbolic pattern-matching approach, similar to that applied to component generation, can be used to automate the process of generating library-specific decomposition methods (or construction methods) for prototypical classes of cells. As defined in the technology compilation algorithm, acquisition templates can be defined to recognize when instances of particular classes of ASIC cells are and are not present in a given cell library. Applicable methods can be expanded to generate decomposition methods that are specific to the recognized cells. Because functional cells can be customized in unpredictable ways, it is not possible to anticipate all ASIC cell classes. Thus, acquisition templates are not intended to cover the entire range of potential ASIC cells; instead, they are intended to recognize commonly occurring or prototypical cell classes.

I validate the effectiveness of the technology compilation algorithm informally with four demonstrations. DTAS is initialized with the generic decomposition methods described in Appendix A, Section A.1, and then stepped through four phases of

```
component ALU_32_4_4_8
    port(I0:      in BIT_VECTOR (31 downto 0);
         I1:      in BIT_VECTOR (31 downto 0);
         ICIN:    in BIT;
         ISEL:    in BIT_VECTOR (3 downto 0);
         O0:      out BIT_VECTOR (31 downto 0);
         OCOUT:   out BIT;
         OREL:    out BIT);
end component;

attribute OPERATIONS of ALU_32_4_4_8: component
  is ( ADD, SUB, INC, DEC, EQ, LT, GT, ZEROP,
       AND, OR, NAND, NOR, XOR, XNOR, LNOT, LIMPL);
```

Figure 9.1: Sample 32-bit/16-function ALU in VHDL.

library upgrades using LOLA.

1. *NAND Implementation*, in which LOLA generates construction methods for mapping Boolean gates into a library containing a single 2-input NAND gate;
2. *Standard Cells*, in which LOLA generates construction methods for mapping n-input Boolean gates into a typical standard-cell library of Boolean cells;
3. *Adders*, in which LOLA generates construction methods for mapping n-bit adders into 2-, 4-, and 16-bit adder cells;
4. *Multiplexers*, in which LOLA generates construction methods for mapping m-by-n multiplexers into 2-to-1, 4-to-1, 4-to-2, and 8-to-4 multiplexer cells.

In each phase, I list the pertinent changes to the cell library, describe the acquisition templates used, and list the methods generated.

I demonstrate how the methods generated by LOLA allow DTAS to maintain the integrity of its design quality with regard to the design of a 32-bit, 16-function ALU, the VHDL specification for which is shown in Figure 9.1. Since the DTAS's generic methods for ALUs make heavy use to both adders and multiplexers, as well as Boolean gates, these library upgrades result in DTAS generating increasingly higher-performance designs. The performance filtering function used in each run of DTAS was the bounded-curve filter used in validating the component decomposition algorithm and defined earlier in Chapter 4, Section 4.5.

The cell sets added to the target ASIC cell library are adapted from LSI Logic's macrocell library (LSI, 1987). In particular, changes are made to LSI's 4-bit adder FA4 eliminating its library-specific inputs and outputs. The 16-bit adder, FA16, is adapted from a full carry look-ahead adder optimized with the MISII (Detjens et al., 1987) logic optimization tool.

9.2 Phase I: NAND Implementation

The first phase initializes the ASIC cell library with a single 2-input NAND gate (ND2). DTAS's generic decomposition methods allow it to to decompose an ALU to the level of generic Boolean gates, including inverters (INV) and 2-input AND, OR, NAND, NOR, XOR, and XNOR gates. Thus, construction methods must be provided that map these basic Boolean gates into NAND implementations using the ND2 cell before DTAS can be used to generate designs for the example ALU.

An example of the sort of acquisition templates needed to bootstrap DTAS into the phase I cell library was seen earlier in Figure 8.9(a). This template generates a method for decomposing an n-input AND gate into a n-input library NAND cell and a generic INV. Four similar acquisition templates are shown in Figure 9.2. Each of these templates is applicable to the phase I cell library, generating methods for mapping INVs, OR, XOR, and XNOR gates into NAND implementations.

The methods generated after expanding the acquisition templates in Figure 9.2 are shown in Figure 9.3; each method is also depicted graphically. Once these methods are added to DTAS' library of decomposition methods, DTAS is able to generate fully-mapped designs for the example 32-bit ALU.

In particular, DTAS generates four alternative ALU designs. The delay versus area characteristics of these designs are compared on the graph shown in Figure 9.4. Design 1 uses an integrated ALU style and ripple-carry adder; design 2 uses an integrated style and ripples two 16-bit carry look-ahead adders; design 3 uses an integrated style and a full 32-bit carry look-ahead adder; each adder has a carry enable input. Design 4 uses a segregated ALU style and a full 32-bit carry look-ahead adder (without carry enable).

```
If library includes
    $cell: NAND(I0^2)
                (O0)
and excludes
   INV(I0)
       (O0)
=>
    acquire
      {
        INV(A)
            (Z)
          ->
              $cell: NAND(<A H>)
                          (Z)
              let H := 1
      }
```

(a)

```
If library includes
    $cell: NAND(I0^$n)
                (O0)
and excludes
   OR (I0^$n)
       (O0)
=>
    acquire
      {
        OR (A^$n)
            (Z)
          ->
              INV(A[0..$n-1])
                  (A.0..$n-1)
              $cell: NAND(A.0..$n-1)
                          (Z)
      }
```

(b)

```
If library includes
    $cell: NAND(I0^2)
                (O0)
and excludes
   XOR (I0^2)
       (O0)
=>
    acquire
      {
        XOR (A^2)
            (Z)
          ->
              INV(A[0])
                  (N.0)
              INV(A[1])
                  (N.1)
              $cell: NAND(A[0] N.1)
                          (I.0)
              $cell: NAND(N.0 A[1])
                          (I.1)
              $cell: NAND(I.0..1)
                          (Z)
      }
```

(c)

```
If library includes
    $cell: NAND(I0^2)
                (O0)
and excludes
   XNOR(I0^2)
       (O0)
=>
    acquire
      {
        XNOR(A^2)
            (Z)
          ->
              INV(A[0])
                  (N.0)
              INV(A[1])
                  (N.1)
              $cell: NAND(A[0..1])
                          (I.0)
              $cell: NAND(N.0..1)
                          (I.1)
              $cell: NAND(I.0..1)
                          (Z)
      }
```

(d)

Figure 9.2: Acquisition templates for NAND implementations: (a) inverter (INV); (b) OR gate; (c) XOR gate; and (d) XNOR gate.

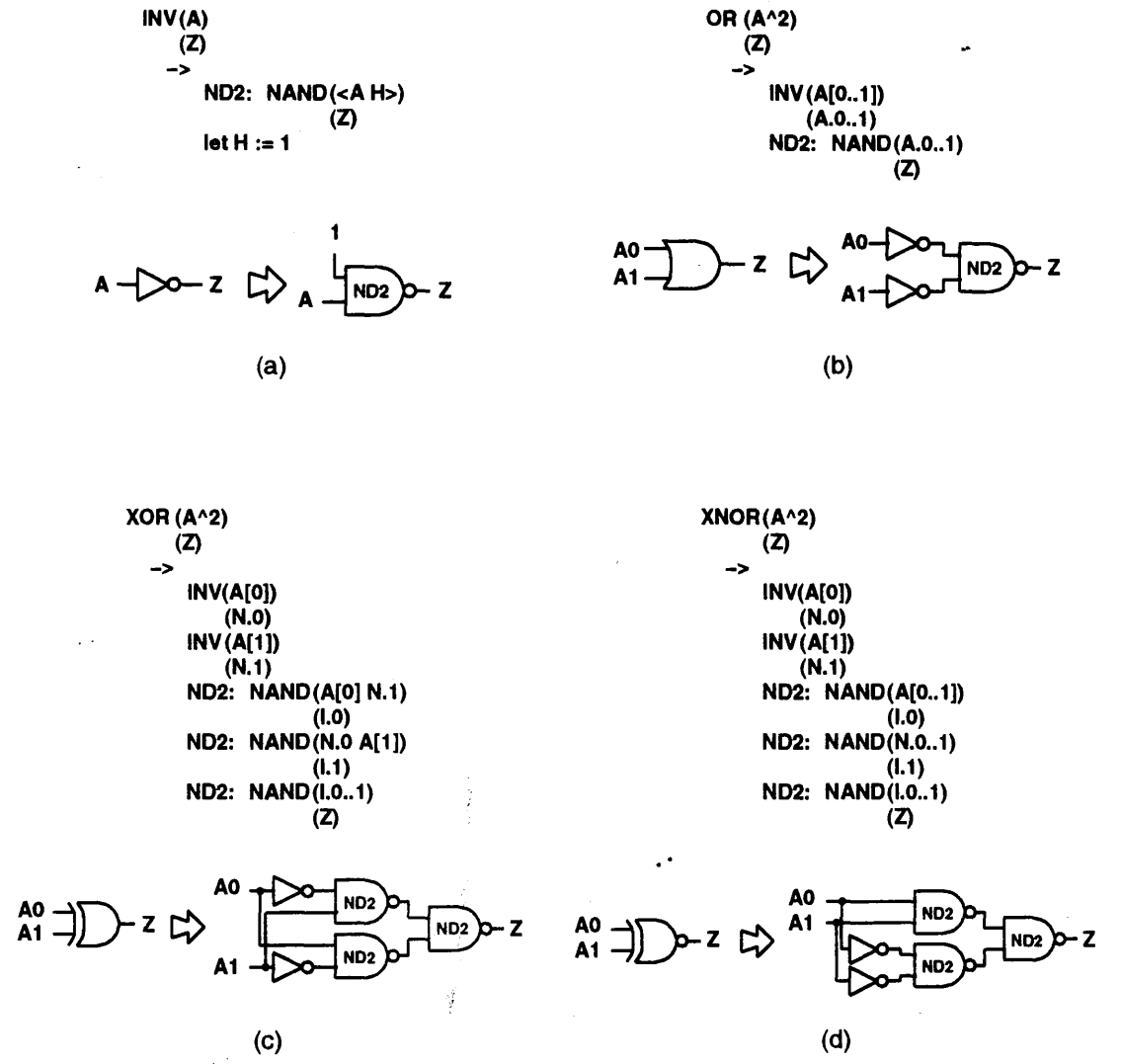


Figure 9.3: Generated methods for: (a) INV; (b) 2-input OR; (c) 2-input XOR; and (d) 2-input XNOR gates using ND2 library cell.

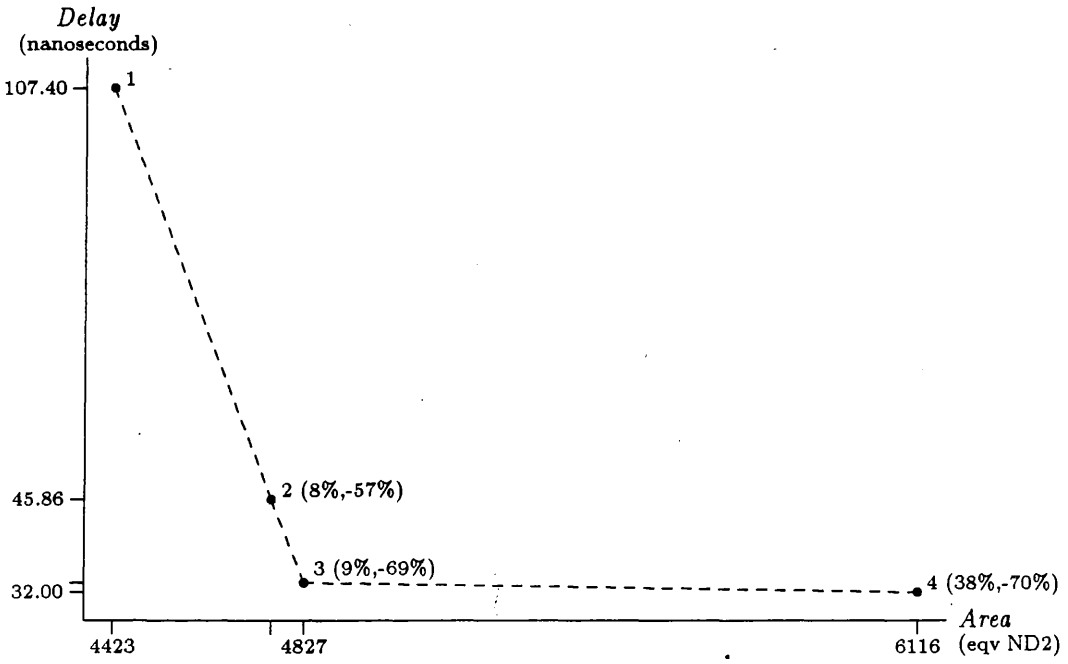


Figure 9.4: Design space for 32-bit/16-function ALU using the phase I cell library.

9.3 Phase II: Standard Cells

In the second phase, the ASIC library is upgraded to contain a variety of Boolean gates typical of standard cell libraries, including two 1-input INVs, 2- and 3-input XOR and XNOR gates, 2-, 3-, and 4-input AND and OR gates, and 2-, 3-, 4-, and 8-input NAND and NOR gates. Wired-ORs and other two-level Boolean cells often found in standard cell libraries are also included in this upgrade, even though they are not used by DTAS.

DTAS's generic methods decompose n-input Boolean gates into configurations of 2-input gates. To take advantage of the phase II cells, DTAS must be provided with construction methods that map n-input gates into the appropriate configuration of 2-, 3-, 4-, and 8-input library cells. Figures 9.5 and 9.6 show two acquisition templates for generating such methods. These templates are explained below.

Given the phase II cell library, there are two types of methods needed. The first type are methods for decomposing components whose width falls in between two cells. For instance, methods for implementing NAND and NOR gates with fewer than 8 but more than 4 inputs. The second type are methods for decomposing components whose width is greater than the largest cell. For instance, methods for implementing NAND and NOR gates with more than 8 inputs, AND and OR gates with more than 4 inputs, and XOR and XNOR gates with more than 3 inputs.

The acquisition template shown in Figure 9.5 addresses the first situation. It generates methods that fill in the gap between one Boolean gate (`$cell1`) and the next smallest (`$cell2`), when the difference in the number of inputs is greater than one. The excludes set tests that there is no Boolean gate of the same type with an intermediate number of inputs. This template is applicable to the phase II cell library, since this library contains a gap between 8 and 4 inputs for its NAND- and NOR-type gates. For instance, this template will match the 8-input NAND gate (ND8) and the 4-input NAND gate (ND4) without also matching the excludes set.

In general, there are two ways to implement a gate with less than 8 inputs and more than 4 inputs. One is with an 8-input gate, placing "identity" signals on unused inputs. The other is with a configuration of smaller gates. Both alternatives must be considered. For instance, implementing a 7-input NAND gate with a 8-input cell might give the best performance, while a 6-input NAND gate might be better implemented out of a two 3-input NAND gates and a 2-input OR gate.

```
if library includes:
    $cell1: $type (I0^$n)
                  (O0)
    $cell2: $type (I0^$m)
                  (O0)
    where member($type, (AND OR NAND NOR XOR XNOR)) && $n > $m+1
  and excludes:
    $type (I0^$w)
          (O0)
    where $n > $w && $w > $m
  =>
     if(member($type, (AND NAND XNOR)))
       let $i := 1
     else
       let $i := 0
     acquire
       {
          $type (A^?n)
                (Z)
            where $n > ?n && ?n > $m
            ->
                $cell1: $type (A[0..?n-1] A.?n..$n-1)
                              (Z)
                for ?i from ?n to $n-1
                  let A.?i := $i
       }
     if($type == NAND)
      let $factor := AND
     else
     if($type == NOR)
      let $factor := OR
     else
      let $factor := $type
     acquire
       {
          $type (A^?n)
                (Z)
            where  $m > ?n && ?n > $n
            ->
                let  ?w := floor(?n/$n)
                     ?r  := mod(?n,$n)
                for ?i from 0 by ?w as ?j from 0 ot $n-?r-1
                   $factor (A[?i..?i+?w-1])
                           (A.?j)
                for ?i from ?w*($n-?r) by ?w+1 as ?j from $n-?r to $n
                   $factor (A[?i..?i+?w])
                           (A.?j)
                $cell2: $type (A.0..$n-1)
                              (Z)
       }
```

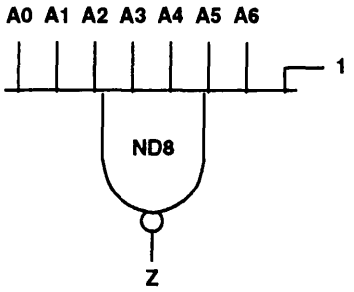
Figure 9.5: Acquisition template for Boolean gates (I).

```
If library includes:
    $cell: $type (I0^$n)
                 (O0)
    where member($type, (AND OR NAND NOR XOR XNOR))
  and excludes:
    $type (I0^$m)
          (O0)
    where $m > $n
=>
    if($type == NAND)
      let $factor    := AND
          $connect   := OR
    else
    if($type == NOR)
      let $factor    := OR
          $connect   := AND
    else
      let $factor    := $type
          $connect   := $type
    acquire
      {
        $type (A^?n)
              (Z)
          where ?n > $n
          ->
              let ?w := floor(?n/$n)
                  ?r  := mod(?n,$n)
              for ?i from 0 by ?w as ?j from 0 to $n-?r-1
                $factor (A[?i..?i+?w-1])
                        (A.?j)
              for ?i from ?w*($n-?r) by ?w+1 as ?j from $n-?r to $n
                $factor (A[?i..?i+?w])
                        (A.?j)
              $cell: $type (A.0..$n-1)
                           (Z)

        $type (A^?n)
              (Z)
          where ?n > $n
          ->
              let ?w := floor(?n/$n)
                  ?r  := mod(?n,$n)
              for ?i from 0 by $n as ?j from 0 to ?w-1
                $cell: $type (A[?i..?i+$n-1])
                             (A.?j)
              if(?r == 0)
                $connect (A.0..?w-1)
                         (Z)
              else
              if(?r == 1)
                $connect (A.0..?w-1 A[?n-1])
                         (Z)
              else
                {
                  $type (A[?n-?r..?n-1])
                        (A.?w)
                  $connect (A.0..?w)
                           (Z)
                }
      }
```

Figure 9.6: Acquisition template for Boolean gates (II).

```
NAND(A^?n)
    (Z)
 where 8 > ?n && ?n > 4
 ->
    ND8:  NAND(A[0..?n] A.?n..7)
                (Z)
    for ?i from ?n to 7
      let A.?i := 1
```

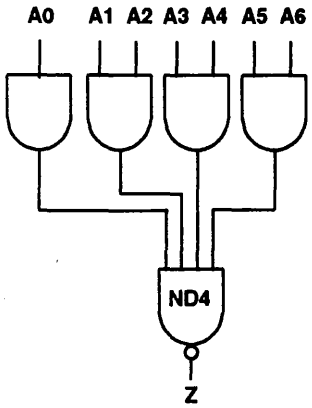
(a)



(b)

```
NAND(A^?n)
    (Z)
 where 8 > ?n && ?n > 4
 ->
    let  ?w := floor(?n/4)
         ?r  := mod(?n,4)
    for ?i from 0 by ?w as ?j from 0 to 3-?r
      AND (A[?i..?i+?w-1])
          (A.?j)
    for ?i from ?w^(3-?r) by ?w+1 as ?j from 4-?r to 3
      AND (A[?i..?i+?w])
          (A.?j)
    ND4:  NAND(A.0..3)
              (Z)
```

(c)



(d)

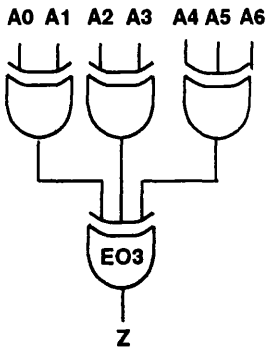
Figure 9.7: Generated methods for Boolean gates (I).

The acquisition template in Figure 9.5 generates methods for both situations. When applied to the 8- and 4-input NAND gates (ND8 and ND4), this template generates the two methods shown in Figures 9.7(a) and (c). When these methods are applied to a 7-input NAND gate, the method in Figure 9.7(a) generates the netlist depicted in Figure 9.7(b), in which the gate is implemented by attaching a high signal to the unused inputs of the 8-input ND8. The method in Figure 9.7(c) generates the netlist depicted in Figure 9.8(d), in which the gate is implemented with the 4-input ND4 and one 1-input and three 2-input generic AND gates.

The acquisition template shown in Figure 9.8 addresses the second situation, i.e., where a gate is required with more inputs than available from the largest library gate. Again, there are two general ways that such a cell can be implemented. One way

```
XOR (A^?n)
    (Z)
 where ?n > 3
 ->
    let  ?w := floor(?n/3)
         ?r  := mod(?n,3)
    for ?i from 0 by ?w as ?j from 0 to 2-?r
        XOR (A[?i..?i+?w-1])
            (A.?j)
    for ?i from ?w*(3-?r) by ?w+1 as ?j from 3-?r to 2
      XOR (A[?i..?i+?w])
          (A.?j)
      EO3:  XOR (A.0..2)
                (Z)
```

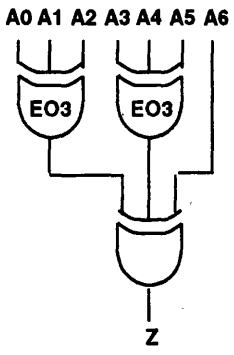
(a)



(b)

```
XOR (A^?n)
    (Z)
 where ?n > 3
 ->
    let  ?w := floor(?n/3)
         ?r  := mod(?n,3)
    for ?i from 0 by 3 as ?j from 0 to ?w-1
      EO3:  XOR (A[?i..?i+2])
                (A.?j)
    if(?r == 0)
      XOR (A.0..?w-1)
          (Z)
    else
    if(?r == 1)
      XOR (A.0..?w-1 A[?n-1])
          (Z)
    else
      {
        XOR (A[?n-?r..?n-1])
            (A.?w)
        XOR (A.0..?w)
            (Z)
      }
```

(c)



(d)

Figure 9.8: Generated methods for Boolean gates (II).

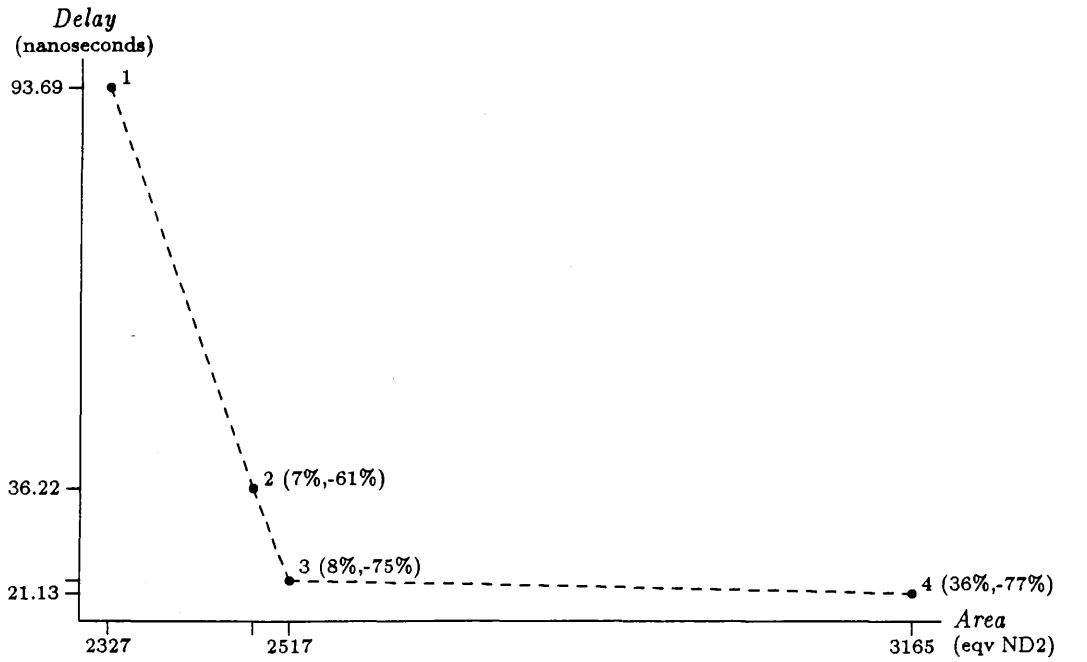


Figure 9.9: Design space for 32-bit/16-function ALU using phase II cell library.

is to factor the required inputs between the available inputs of the library cell, where NAND gates factored between AND gates, NOR gates between OR gates, and other gates between gates of the same type. The other way is to divide the desired inputs between multiple instances of the library cell and connect their outputs, where NAND gates must be connected by OR gates, NOR gates by AND gates, and other gates by gates of the same type.

The acquisition template in Figure 9.6 generates methods for both situations. When this template is applied to a cell such as the 3-input XOR (EO3) from the phase II cell library, it generates the two alternative methods shown in Figures 9.8(a) and (c). When these methods are applied to a 7-input XOR, they generate the netlist implementations depicted graphically in Figures 9.8(b) and (d), respectively.

Once all methods have been generate for the cells in the phase II library, DTAS generates four designs for the example ALU. The delay versus area characteristics of these designs are compared on the graph shown in Figure 9.9. These four designs use the same styles found in the four designs generated with the phase I cell library. However, compared to the graph seen in Figure 9.4, this one library upgrade significantly improves design quality.

9.4 Phase III: Adders

In the third phase, the ASIC library is upgraded to contain adders of varying widths, including a half adder, two 1-bit adders, a 2-bit adder, a 4-bit adder, and a 16-bit adder. Half adders and 1-bit full adders are not uncommon in standard-cell libraries, 2-bit adders are less common, 4-bit adders and 16-bit adders are more typical of macro- and megacell libraries.

As explained earlier, the 4-bit and 16-bit adders used in this upgrade are adaptations of the adders actually described in the LSI Logic's macrocell databook; the adaptations eliminate atypical, library-specific inputs and outputs. In general, decomposition methods for atypical cells need to be generated by hand. Acquisition templates are intended for automatically generating "easy" methods for cell types that can be anticipated or that are common across many libraries.

DTAS's generic methods already decompose n-bit adders into 1-bit adders, so no library-specific methods are needed for the two 1-bit adder cells. Also, because there are 1-bit adders in this library, no methods are needed for implementing a full adder with two half adders. Methods are needed for mapping into the 2-, 4-, and 16-bit adder cells.

Although logically more complex than the Boolean gates of the phase II cell library, the library-specific methods needed to take advantage of the 2-, 4-, and 16-bit adder cells are quite similar. The difference in their specifications is that prototypical adders have two n-bit inputs, an n-bit output, and a carry input and output, as opposed to a single n-bit input and a 1-bit output. Although these differences must be reflected in the acquisition templates for adders, the templates themselves have the same general structure and function of the templates for Boolean gates seen earlier.

In particular, for the phase III cell library decomposition methods need to be generated that handle three situations: when an n-bit adder is required

1. where n is less than 16 but greater than 4;
2. where n is equal to 3 (i.e., between the 4- and 2-bit library adders);
3. where n is greater than 16.

These methods can be generated by the acquisition templates shown in Figures 9.10 and 9.11.

The acquisition template in Figure 9.10 generates methods that address the first two situations. This template is applicable when there are two full adders in the cell library, when the data widths of these cells differs by more than one, and when there is no other library cell with an intermediate data width. There are two `acquire`

```
If library includes:
    $cell1: ADD (I0^$n I1^$n C.0)
                (O0^$n C.?n)
    $cell2: ADD (I0^$m I1^$m C.0)
                (O0^$m C.?n)
    where $n > 1 && $m > $n+1
  and excludes:
    ADD (I0^$w I1^$w C.0)
        (O0^$w C.?n)
    where  $m > $w && $w > $n
=>
    if($m/$n > 2)
      acquire
        {
          ADD (A^?n B^?n C.0)
              (S^?n C.?n)
            where $m > ?n && ?n > $n
            ->
                let ?r  := mod(?n,$n)
                for ?i from 0 to ?n-?r-1 by $n
                  $cell1:  ADD (A[?i..?i+$n-1] B[?i..?i+$n-1] C.?i)
                               (S[?i..?i+$n-1] C.?i+$n)
                if(?r > 0)
                  ADD (A[?n-?r-1..?n-1] B[?n-?r-1..?n-1] C.?n-?r-1)
                      (S[?n-?r-1..?n-1] C.?n)
        }
    else
      acquire
        {
          ADD (A^?n B^?n C.0)
              (S^?n C.?n)
            where $m > ?n && ?n > $n
            ->
                let ?r  := mod(?n,$n)
                $cell1:  ADD (A[0..$n-1] B[0..$n-1] C.0)
                             (S[0..$n-1] C.$n)
                ADD (A[?n-?r-1..?n-1] B[?n-?r-1..?n-1] C.?n-?r-1)
                    (S[?n-?r-1..?n-1] C.?n)
        }
```

Figure 9.10: Acquisition templates for adders (I).

```
If library includes:
    $cell:  ADD (I0^$n I1^$n C.0)
                (O0^$n C.?n)
    where  $n > 1
and excludes:
    ADD (I0^$w I1^$w C.0)
        (O0^$w C.?n)
    where  $w > $n
=>
    acquire
      {
        ADD (A^?n B^?n C.0)
            (S^?n C.?n)
          where ?n > $n
          ->
              let ?r  := mod(?n,$n)
              for ?i from 0 to ?n-?r-1 by $n
                $cell:  ADD (A[?i..?i+$n-1] B[?i..?i+$n-1] C.?i)
                            (S[?i..?i+$n-1] C.?i+$n)
              if(?r > 0)
                ADD (A[?n-?r-1..?n-1] B[?n-?r-1..?n-1] C.?n-?r-1)
                    (S[?n-?r-1..?n-1] C.?n)
      }
```

Figure 9.11: Acquisition templates for adders (II).

actions in the body of this template. The first handles the situation where the data width of the smaller cell (`$cell1`) is less than the larger (`$cell2`) by a factor of more than two. This condition indicates that several of the smaller cell may have to be cascaded to implement an n-bit adder, where n is close to the width of the larger cell. When the difference between the two adder cells is less than a factor of two, then the n-bit adder can be implemented with a single library adder plus a smaller generic adder handling the remaining inputs.

The template in Figure 9.10 is applicable to the 16-bit and 4-bit adder cells (FA16 and FA4) of the phase III library, as well as the 4-bit and 2-bit adder cells (FA2). When applied in both instances, it generates the two decomposition methods shown in Figure 9.12(a), respectively.

The acquisition template in Figure 9.11 generates methods for the third situation described above, i.e., for n-bit adders where n is greater than the data width of the largest library adder. This template generates a method that implements an n-bit adder by rippling as many of the large library adders as needed to cover no more than n bits and then adding a smaller generic adder to cover the remaining inputs.

The template in Figure 9.11 is applicable to the 16-bit adder cell (FA16) from the phase III library. When expanded, it generates the method shown in Figure 9.12(b).

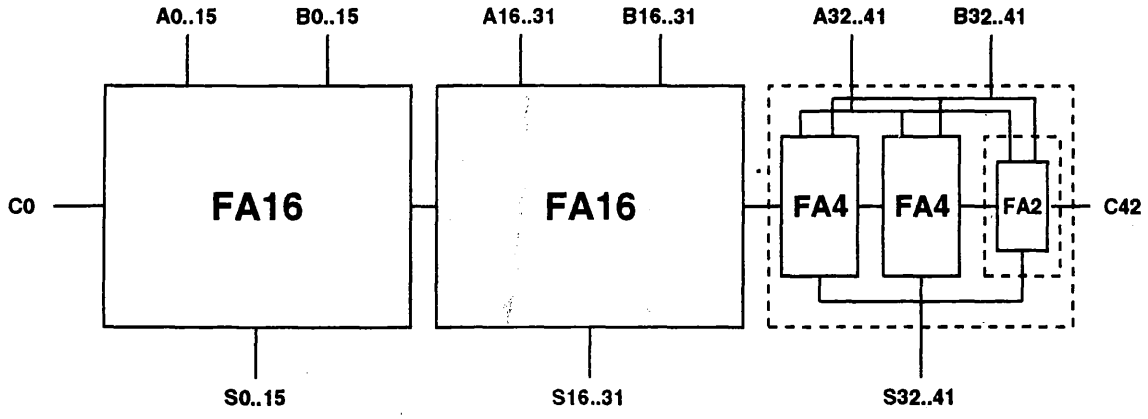
```
ADD (A^?n B^?n C.0)
      (S^?n C.?n)
   where 16 > ?n && ?n > 4
   ->
      let ?r := mod(?n,4)
      for ?I from 0 to ?n-?r-1 by 4
         FA4:  ADD (A[?I..?I+3] B[?I..?I+3] C.?I)
                   (S[?I..?I+3] C.?I+4)
      if(?r > 0)
         ADD (A[?n-?r-1..?n-1] B[?n-?r-1..?n-1] C.?n-?r-1)
             (S[?n-?r-1..?n-1] C.?n-1)
```

```
ADD (A^?n B^?n C.0)
      (S^?n C.?n)
   where 4 > ?n && ?n > 2
   ->
      let ?r := mod(?n,2)
      FA2:  ADD (A[0..1] B[0..1] C.0)
                (S[0..1] C.2)
      ADD (A[?n-?r-1..?n-1] B[?n-?r-1..?n-1] C.?n-?r-1)
          (S[?n-?r-1..?n-1] C.?n-1)
```

(a)

```
ADD (A^?n B^?n C.0)
      (S^?n C.?n)
   where ?n > 16
   ->
      let ?r  := mod(?n,16)
      for ?I from 0 to ?n-?r-1 by 16
         FA16:  ADD (A[?I..?I+15] B[?I..?I+15] C.?I)
                    (S[?I..?I+15] C.?I+16)
      if(?r > 0)
         ADD (A[?n-?r-1..?n-1] B[?n-?r-1..?n-1] C.?n-?r-1)
             (S[?n-?r-1..?n-1] C.?n)
```

(b)



(c)

Figure 9.12: Generated methods for adders.

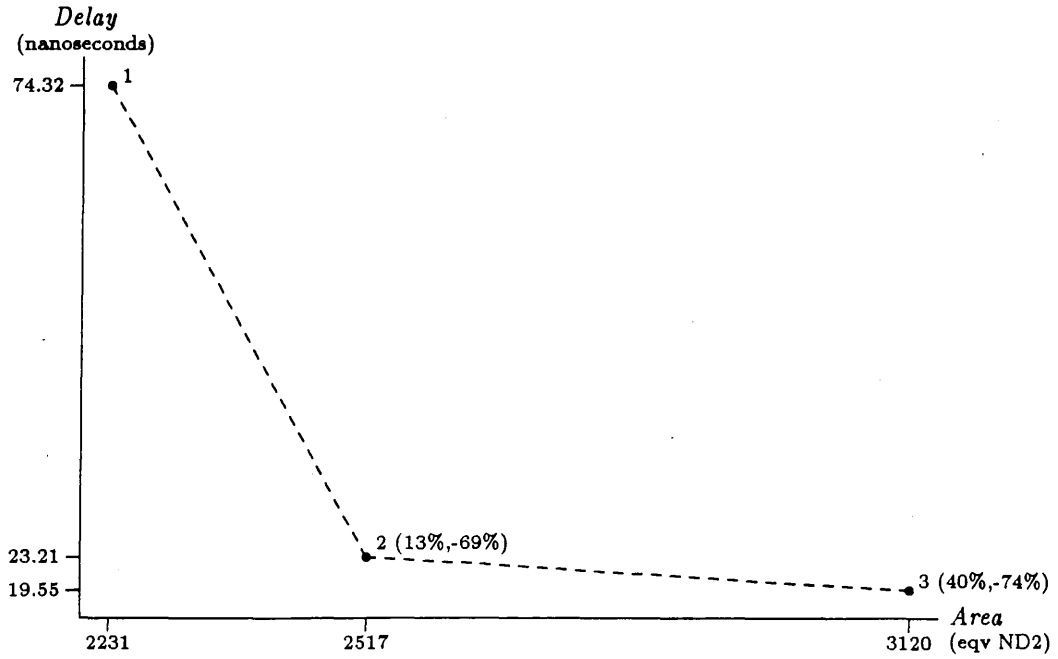


Figure 9.13: Design space for 32-bit/16-function ALU using phase III cell library.

The three methods shown in Figures 9.12(a) and (b) can be used together. For instance, if DTAS is now asked to generate a 42-bit adder, it would find one implementation using the 16-, 4-, and 2-bit library adders, as depicted graphically in Figure 9.12(c).

Given these three methods, DTAS generates the three designs for the example ALU that are plotted in the delay verses area graph shown in Figure 9.13. When compared to the designs generated for the phase II cell library (Figure 9.9), DTAS is able to improve over the first design by using 32 1-bit adder cells for the integrated style ALU with a ripple-carry adder and to improve over the last design by using two 16-bit adder cells for segregated style ALU. Because the integrated style ALU requires an adder with a carry enable input, design 2 still uses a carry look-ahead adder implemented with Boolean gates.

9.5 Phase IV: Multiplexers

In the fourth and final phase, the ASIC library is upgraded to contain several multiplexers, including 2-to-1, 4-to-1, 4-to-2, and 8-to-4 multiplexer cells, named MUX21H, MUX41H, MUX42H, and MUX84H, respectively. The difference between this upgraded and the earlier ones is that multiplexers have a control port, which represents their select input; the variations on the functionality of the control port must be figured into the acquisition templates.

In DTAS's generic decomposition methods, multiplexers are defined as bitwise components, i.e., as having m n-bit inputs. A control port SEL selects between one of the m inputs, which is directed to a single n-bit output. Thus, the 4-to-1 multiplexer (MUX41H) from the phase IV cell library has four 1-bit inputs and a 2-bit binary control port, while the 8-to-4 multiplexer (MUX84H) has two 4-bit inputs and a 1-bit binary control port. In generating library-specific methods for multiplexers, it is necessary to generate methods that generalize over n, m, and the style of the control port. These tasks are exemplified by the four acquisition templates shown in Figures 9.14, 9.15, 9.17, and 9.19.

The two acquisition templates in Figures 9.14 and 9.15 generalize n, the number of bits in each of the m inputs of a multiplexer. The template in Figure 9.14 is applicable when the cell library contains two multiplexers whose data widths differ by more than one and there are no multiplexers of an intermediate width. This situation exists in the phase IV library with the 4-to-2 multiplexer (MUX42H) and the 8-to-4 multiplexer (MUX84H), for which the template will generate a library-specific method for a 6-to-3 multiplexer.

The other template, in Figure 9.15, is applicable to the m-input multiplexer (for each distinct m) with the largest n, such as the 4-to-1 multiplexer (MUX41H), for which $m = 4$, and the 8-to-4 multiplexer (MUX84H), for which $m = 2$. It generates methods for multiplexers with data widths larger than n. For instance, when this template is applied to MUX84H, it generates the method shown in Figure 9.16(a). The netlist generated by this method when used to implement a 20-to-10 multiplexer is depicted in Figure 9.16(b).

The acquisition template in Figure 9.17 generalizes m, the number of n-bit inputs of a multiplexer. This particular template generates methods for implementing multiplexers with larger m's. This template is also constrained to multiplexers with binary control ports; there are other, slightly more complex templates, for multiplexers with unary control ports. This template is applicable to n-bit multiplexers for which there is no larger m, such as the 4-to-1 multiplexer (MUX41H), for which $n = 1$, the 4-to-2 multiplexer (MUX42H), for which $n = 2$, and the 8-to-4 multiplexer (MUX84H), for which $n = 4$.

```
if library includes:
    $cell1: MUX(I0^$m:$n1 SEL^$s)
                (O0^$n1)
              :ctrl SEL $attrs
    $cell2: MUX(I0^$m:$n2 SEL^$s)
                (O0^$n2)
              :ctrl SEL $attrs
    where $n2 > $n1 && $n1 > 1
  and excludes:
    MUX(I0^$m:$n3 SEL^$s)
        (O0^$n3)
      :ctrl SEL $attrs
    where $n2 > $n3 && $n3 > $n1
  =>
     if($n2/$n1 > 2)
       acquire
         {
           MUX(A^$m:?n SEL^$s)
               (Z^?n)
             :ctrl SEL $attrs
             where $n2 > ?n && ?n > $n1
             ->
                 let ?r := mod(?n,$n1)
                 for ?i from 0 to ?n-?r-1 by $n1
                   $cell1: MUX(A[0..$m-1:?i..?i+$n1-1] SEL^$s)
                               (Z[?i..?i+$n1-1])
                 if(?r > 0)
                   MUX(A[0..$m:?n-?r-1..?n-1] SEL^$s)
                       (Z[?n-?r-1..?n-1])
                     :ctrl SEL $attrs
         }
     else
       acquire
         {
           MUX(A^$m:?n SEL^$s)
               (Z^?n)
             :ctrl SEL $attrs
             where $n2 > ?n && ?n > $n1
             ->
                 let ?r := mod(?n,$n1)
                 $cell1: MUX(A[0..$m-1:0..?$n1-1] SEL^$s)
                             (Z[0..$n1-1])
                 MUX(A[0..$m:?n-?r-1..?n-1] SEL^$s)
                     (Z[?n-?r-1..?n-1])
                   :ctrl SEL $attrs
         }
```

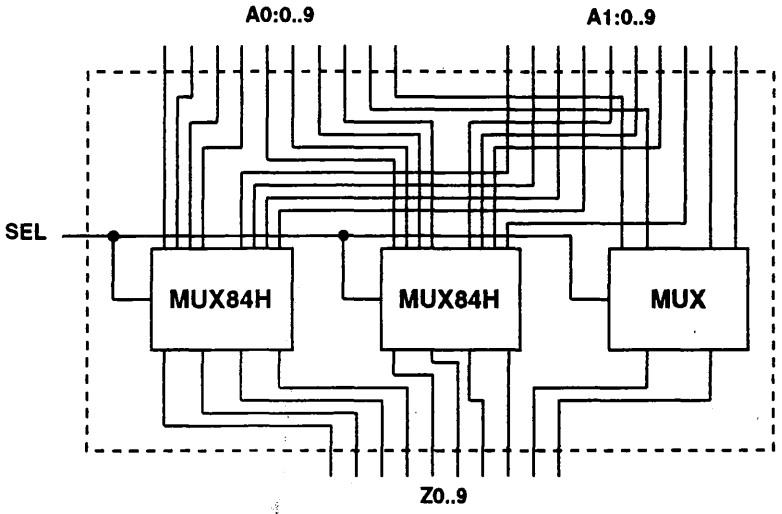
Figure 9.14: Acquisition templates for multiplexers (I).

```
if library includes:
    $cell: MUX(I0^$m:$n SEL^$s)
               (O0^$n)
             :ctrl SEL $attrs
    where $n > 1
and excludes:
    MUX(I0^$m:$w SEL^$s)
       (O0^$w)
     :ctrl SEL $attrs
    where $w > $n
=>
    acquire
      {
        MUX(A^$m:?m SEL^s)
           (Z^?n)
         :ctrl SEL $attrs
         where ?n > $n
         ->
             let ?r := mod(?n,$n)
             for ?i from 0 to ?n-?r-1 by $n
                $cell: MUX(A[0..$m-1:?i..?i+$n-1] SEL^$s)
                           (Z[?i..?i+$n-1])
             if(?r > 0)
                MUX(A[0..$m:?n-?r-1..?n-1] SEL^$s)
                   (Z[?n-?r-1..?n-1])
                 :ctrl SEL $attrs
      }
```

Figure 9.15: Acquisition templates for multiplexers (II).

```
MUX (A^2:?n SEL)
    (Z^?n)
  :ctrl SEL (:keys (0 1) :style BINARY)
  where ?n > 4
  ->
      let ?r := mod(?n,4)
      for ?i from 0 to ?n-?r-1 by 4
         MUX84H:   MUX (A[0..1:?i..?i+3] SEL)
                       (Z[?i..?i+3])
      if(?r > 0)
         MUX (A[0..1:?n-?r-1..?n-1] SEL)
             (Z[?n-?r-1..?n-1])
           :ctrl SEL (:keys (0 1) :style BINARY)
```

(a)



(b)

Figure 9.16: Generated methods for multiplexers (I).

```
if library includes:
    $cell: MUX(I0^$m:$n SEL^$s1)
               (O0^$n)
           :ctrl SEL (:style BINARY $$attrs)
           :select (0..$m-1)
    where $m = 2^$s1
  and excludes:
    MUX(I0^$w:$n SEL^$s2)
        (O0^$n)
      :ctrl SEL (:style BINARY $$)
      :select (0..$w-1)
    where $w = 2^$s2 && $w > $m
  =>
     acquire
       {
         MUX(A^?m:$n SEL^?s)
             (Z^$n)
           :ctrl SEL (:style BINARY)
           :select (0..?m-1)
           where ?m > $m && mod(?m,$m) = 0
           ->
               let ?w := ?m/$m
               for ?i from 0 by $m as ?j from 0 to ?w-1
                  $cell: MUX(A[?i..?i+$m-1:0..$n-1] SEL[0..$s1-1])
                             (Z.?j.0..$n-1)
               MUX(Z.0..?w-1.0..$n-1 SEL[$s1..?s-1])
                    (Z[0..$n-1])
                  :ctrl SEL (:style BINARY)
                  :select (0..?w-1)
       }
```

Figure 9.17: Acquisition templates for multiplexers (III).

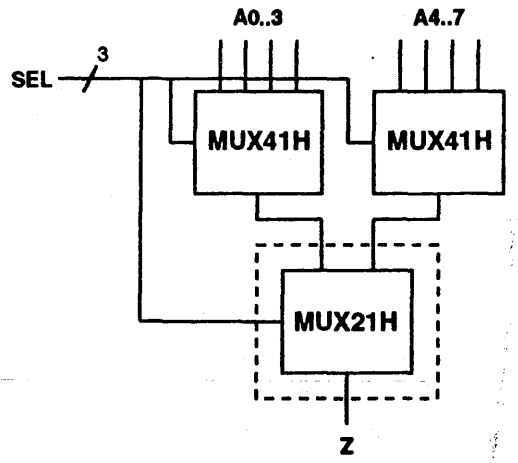
When applied to MUX41H, the template in Figure 9.17 generates the method shown in Figure 9.18(a). When applied to MUX84H, it generates the method shown in Figure 9.18(c). The netlist generated by the first method when used to implement an 8-to-1 multiplexer is depicted in Figure 9.18(b), while the netlist generated by the second method when used to implement a 32-to-4 multiplexer is depicted in Figure 9.18(d).

```
MUX(A^?m SEL^?s)
    (Z)
  :ctrl SEL (:style BINARY)
  :select (0..?m-1)
  where ?m > 4 && mod(?m,4) = 0
  ->
     let ?w := ?m/4
     for ?i from 0 by 4 as ?j from 0 to ?w-1
        MUX41H:   MUX(A[?i..?i+3] SEL[0..1])
                      (Z.?j)
     MUX(Z.0..?w-1 SEL[2..?s-1])
         (Z)
        :ctrl SEL (:style BINARY)
        :select (0..?w-1)
```

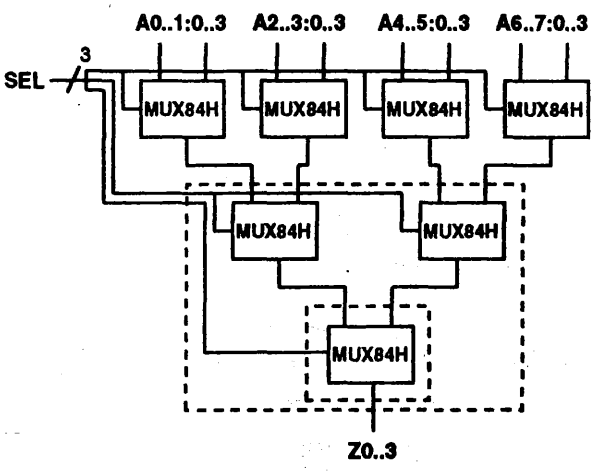
(a)

```
MUX(A^?m:4 SEL^?s)
    (Z^?4)
  :ctrl SEL (:style BINARY)
  :select (0..?m-1)
  where ?m > 2 && mod(?m,2) = 0
  ->
     let ?w := ?m/2
     for ?i from 0 by 2 as ?j from 0 to ?w-1
        MUX84H:   MUX(A[?i..?i+1:0..3] SEL[0])
                      (Z.?j.0..3)
     MUX(Z.0..?w-1:0..3 SEL[1..?s-1])
         (Z[0..3])
        :ctrl SEL (:style BINARY)
        :select (0..?w-1)
```

(c)



(b)



(d)

Figure 9.18: Generated methods for multiplexers (II).

```
if library excludes:
    MUX(I0^$m:$n SEL^$s)
        (O0^$n)
      :ctrl SEL (:style UNARY $$attrs)
  =>
      acquire
        {
          MUX(A^?m:?n SEL^?m)
              (Z^n)
            :ctrl SEL (:style BINARY)
            :select (0..?m-1)
            where integerp(log(?m,2))
            ->
                let ?s := log(?m,2)
                ENCODE(SEL[0..?m-1])
                        (S[0..?s-1])
                MUX(A[0..?m-1:0..?n-1] SEL=>S[0..?s-1])
                    (Z[0..?n-1])
                  :ctrl SEL (:style BINARY)
                  :select (0..?m-1)
        }
```

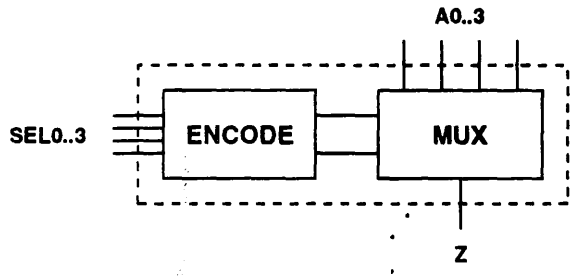
Figure 9.19: Acquisition templates for multiplexers (IV).

Finally, the acquisition template in Figure 9.19 generalizes the style of the control port. In particular, this template generates a method for implementing a multiplexer with a unary control port, such as the multiplexers defined in the GENUS library, with an encoder and a binary-controlled multiplexer. This template is only applicable when the cell library contains no unary-controlled multiplexer. The method generated by this template is shown in Figure 9.20(a). When this method is applied to a unary-controlled 4-to-1 multiplexer, it generates the netlist depicted in Figure 9.20(b).

Given these new methods, DTAS generates the three designs for the example ALU that are plotted in the delay versus area graph shown in Figure 9.21. When compared to the designs generated for the phase III cell library (Figure 9.13), DTAS is only able to improve over the last design. This is because only the segregated style ALU uses a multiplexer. Although the multiplexer cells do little to change the delay through the segregated style ALU, they do reduce its area.

```
MUX (A^?m:?n SEL^?m)
      (Z^?n)
   :ctrl SEL (:style UNARY)
   :select (0..?m-1)
   where integerp(log(?m,2))
   ->
        let ?s := log(?m,2)
        ENCODE(SEL[0..?m-1])
                (S[0..?s-1])
        MUX (A[0..?m-1:0..?n-1] SEL=>S[0..?s-1])
              (Z[0..?n-1])
           :ctrl SEL (:style BINARY)
           :select (0..?m-1)
```

(a)



(b)

Figure 9.20: Generated methods for multiplexers (III).

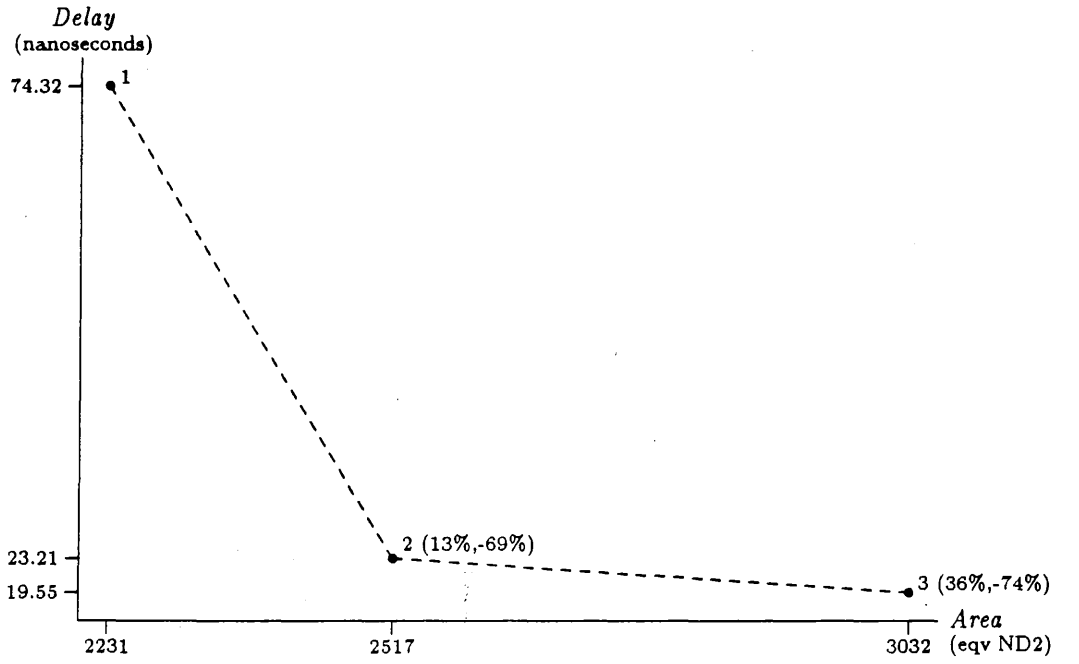


Figure 9.21: Design space for 32-bit/16-function ALU using phase IV cell library.

9.6 Summary

I have claimed that the technology compilation algorithm, as implemented in LOLA, can be used as an approach to automating technology adaptation with respect to the component decomposition algorithm, as implemented in DTAS. I have validate the effectiveness of the technology compilation algorithm informally by demonstrating how it maintains DTAS's design knowledge as the target ASIC cell library undergoes four upgrades.

1. The first phase initialized the cell library with a 2-input NAND gate.
2. The second phase added Boolean gates typical of standard-cell libraries.
3. The third phase upgraded the cell library with adder cells, including 2-, 4-, and 16-bit adders.
4. The fourth phase upgraded the cell library with multiplexer cells, including 2-to-1, 4-to-1, 4-to-2, and 8-to-4 multiplexers.

After applying LOLA to each library upgrade, I demonstrated the resulting methods by applying DTAS to the design of a 32-bit/16-function ALU. The cumulative results across all four library upgrades are plotted in the delay versus area graph shown in Figure 9.22. The set of designs generated by DTAS after each upgrade are connected by dashed lines. Design 1 of each set is labeled by the upgrade phase, as are the last designs from the second, third, and fourth upgrade phase. This graph shows how the inclusion of more complex library cells successively improves DTAS's design quality.

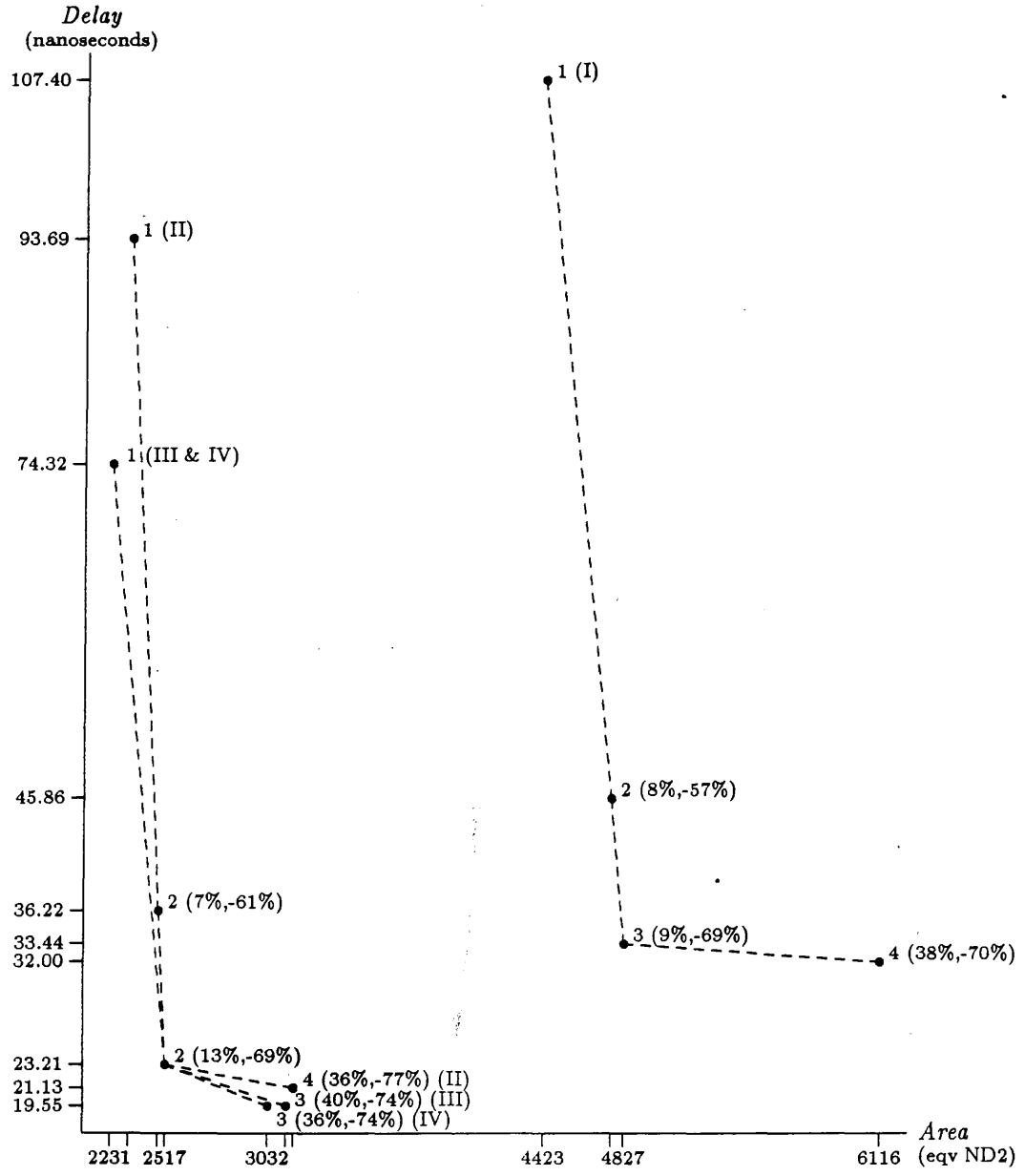


Figure 9.22: Cumulative design space for 32-bit/16-function ALU.

Chapter 10
Conclusion

10.1 Summary of Dissertation

In this dissertation, I have described a symbolic pattern-matching approach to component generation and, relative to this, an approach to automating technology adaptation. I have defined the component decomposition algorithm and technology compilation algorithm that formalize these two approaches and have described implementations of each, in the DTAS component generation system and the LOLA technology adaptation system, respectively. I have presented empirical results to validate the utility of my approach to component generation and have presented a demonstration to validate my approach to technology adaptation.

The approach to component generation that I define in the component decomposition algorithm has two significant benefits. First, it enables the use of complex functional library cells, such as multiplers, adders, and ALUs, in the generation of designs for generic functional components. Second, it effectively searches the design space for designs that make desirable trade-offs between design constraints, such as area and delay. To validate these claims, I have shown the results of four sets of experiments using the DTAS component generation system.

The first set of experiments show how the search control principles of the component decomposition algorithm allowed DTAS to find desirable designs from design spaces that were computationally intractable to enumerate. The second set show how the encapsulation of design styles in decomposition methods allowed DTAS to compare design alternatives and find ranges of designs that make desirable trade-offs between area and delay. The third set of experiments show how the design generated by DTAS compared to the same designs when passed through the MISII logic optimizer; DTAS's designs were often close in performance to designs generated with MISII. The fourth set show how the use of functional decomposition and functional specification allowed DTAS to map designs into libraries of complex functional cells and generate higher-performance designs than was possible when mapping designs into a library of simple Boolean cells.

The approach to technology adaptation that I define in the technology compilation algorithm is significant because it bootstraps the component decomposition algorithm into new ASIC cell libraries, as well as cell libraries as they undergo upgrades. In this way, the technology compilation algorithm automates the task of maintaining technology independence. As defined in the technology compilation algorithm, acquisition templates can be written that recognize classes of commonly occuring ASIC cells and that generate appropriate library-specific decomposition methods when instances of these classes do or do not appear in a given cell library. To validate this claim, I have demonstrated the application of the LOLA technology adaptation system to a cell library as it undergoes four phases of evolution.

10.2 Summary of Contributions

The research describe in this dissertation makes three fundamental contributions. First, it shows an approach to component generate that can map designs into complex functional layout cells. Second, it shows an approach to effectively searching the space of design alternatives. Third, it shows an approach to automating technology adaptation in component generation.

Current approaches to component generation are unable to map designs into complex functional library cells. The use of such cells can improve design quality. By using an abstract symbol language to define the functionality of ASIC library cells, I am to able represent complex cells as succinctly as simple cells. This succinct representation allows my approach to component generation to map RT components into configurations of complex cells and to avoid the computational complexity of Boolean subgraph matching that hinders other approaches to component generation.

Current approaches to component generation do little or no search between alternative design styles; design styles are selected on the basis of design constraints. Using alternative design styles and mixing design styles can lead to designs that make desirable trade-offs between design constraints. By extending the symbol language used to define the functionality of RT components and library cells into a pattern language, I am able to define decomposition methods that implement individual levels of functional decomposition. Such methods encapsulate design styles. By using a branch-and-bound search strategy, my approach to component generation can dynamically explore the space of design alternatives and find mixes of design styles that make desirable trade-offs between constraints such as area and delay.

The desire to maintain technology independence has restricted previous approaches to component generation to using Boolean logic and graph-matching approaches, which are incapable of taking advantage of complex functional library cells.

My approach to component generation shows that technology-specific design knowledge can be used to map designs into complex cells and, thus, to improve design quality. My approach to technology adaptation shows that the maintenance of technology independence can be automated, at least partially, within this framework.

10.3 Status

The DTAS component generation system and LOLA technology adaptation system have been implemented in Common Lisp on Sun-3 and Spark workstations. DTAS can design a wide range of combinational components and some sequential components, i.e., latches and flip flops, registers, and up/down counters. LOLA can generate methods for using library encoders and decoders, as well as Boolean gates, adders, and multiplexers.

The space of combinatorial and sequential components that a component generation should be capable of handling is depicted in Figure 10.1. The scope of DTAS's capabilities include the component types within the shaded area of this space. DTAS is only capable of designing one style of comparator and a subset of shifters and counters, so these types are only partially included in the shaded region. The scope of LOLA's capabilities is smaller than DTAS's and is limited to the component types within the darkly shaded region.

It would be straightforward to add sufficient decomposition methods that expand the capabilities of DTAS to include a large subset of the entire space of components depicted in Figure 10.1. DTAS's algorithms for computing delay, for extracting Boolean equations hierarchical designs, and for simulating design behavior from Boolean equations currently only handle combinatorial components with no feedback loops and would have to be modified. Acquisition templates can also be added to LOLA to extend its capabilities for both combinatorial and sequential prototypical cell types.

10.4 Future Directions

At a larger level, there are several directions in which this dissertation research can drive future efforts. One direction is to explore the impact that a DTAS-like component generation system has on high-level synthesis. Until recently, approaches to high-level synthesis made simplifying assumptions about the RT components into which they were mapping designs. How do these approaches scale up now that DTAS

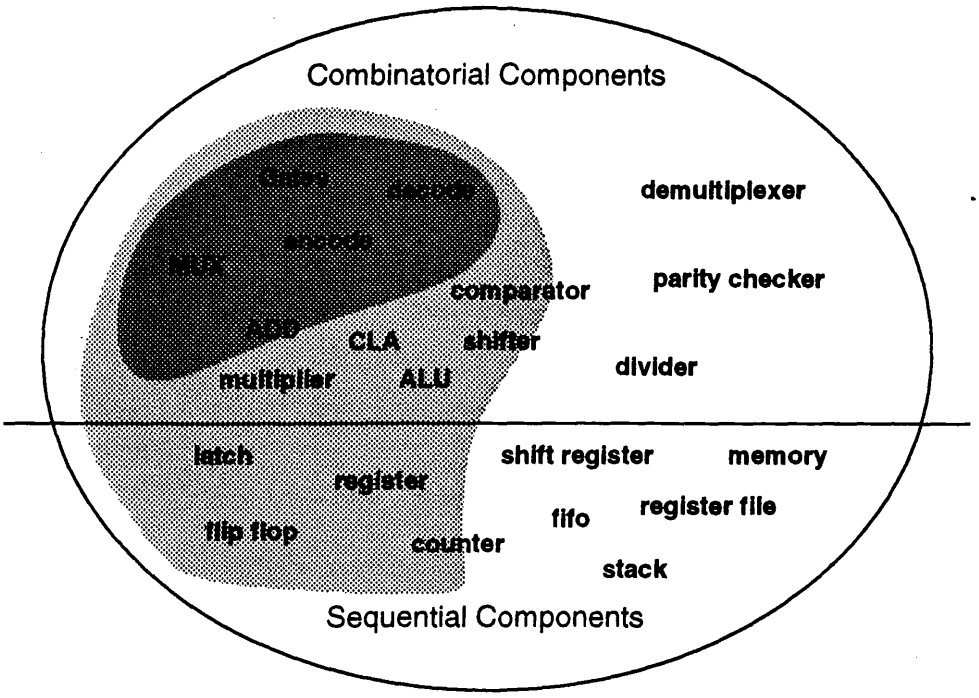


Figure 10.1: Space of components: (shaded region) what DTAS can design; and (darkly shaded region) cell types LOLA can recognize.

can generate technology-specific data on a wide range of RT components? Can new approaches be developed that take advantage of this data to improve design quality in high-level synthesis?

Likewise, a second direction is to explore the impact that a DTAS-like tool has on layout synthesis. Now that there is a synthesis tool that can take advantage of functional layout cells, it is desirable to extend the level of complexity of these cells. How does this effect current approaches to layout synthesis.

A third direction of research is to optimize the designs generated by a system such as DTAS. There are actually two levels at which optimization can take place. First, there are optimizations on the cell level, i.e., the leaves of a hierarchical netlist. For instance, how can logic optimization be improved by functionally partitioning large designs of Boolean gates and passing individual partitions through conventional logic synthesis tools? Extending this, how can conventional logic synthesis techniques be adapted to optimize networks that contain complex functional cells? Second, there are optimizations on the hierarchical structural netlist itself. Can optimizations be

performed at intermediate levels that provide preformance improvements that are computationally intractable at the cell level? Can further optimizations be made between levels?

A final direction of research is to find an approach to acquiring decomposition methods for atypical library cells for which there are no appropriate acquisition templates. I do not believe that yet another level of templates is the answer. However, it may be possible to develop an automated assistant for generating acquisition templates using fundamental principles of digital design and a great deal of user interaction.

Appendix A
The SYN Files

As noted in Chapter 6, component types and decomposition methods are organized in SYN files. This appendix contains a complete listing of the DTAS SYN files, generic, GENUS specific, and cell library specific.

A.1 Generic SYN Files

This section lists DTAS's generic base of component types and methods.

A.1.1 GATES.SYN

The file GATES.SYN defines generic Boolean gates, including the function types: AND, OR, NAND, NOR, XOR (exclusive-OR), XNOR (exclusive-NOR or equivalence), INV (inverter), BUF (buffer), and GATE (for other Boolean gates, such as wired-ORs). With the exception of INV, BUF, and GATE, all gates are defined as "bitwise" components, i.e., they have m n-bit inputs and out n-bit output. INVs and BUFs are defined as having one n-bit input and one n-bit output.

```
/* GATES.SYN */

COMPONENT TYPES:

AND(I^?m:?n)
   (O^?n)
  :data I[] O[]

OR(I^?m:?n)
  (O^?n)
  :data I[] O[]

NAND(I^?m:?n)
    (O^?n)
  :data I[] O[]

NOR(I^?m:?n)
   (O^?n)
  :data I[] O[]

XOR(I^?m:?n)
   (O^?n)
  :data I[] O[]

XNOR(I^?m:?n)
    (O^?n)
  :data I[] O[]

INV(I^?n)
   (O^?n)
  :data I[] O[]

GATE(I^?n)
    (O)
  :data I[] O

DESIGN RULES:

/* Primitive Logic Gates */

INV(I^?n)
   (O^?n)
 where ?n > 1
 -> for ?i from 1 to ?n
      let O[?i] := I[?i]';

AND(I)
   (O)
 -> let O := I;
```

```
AND(I^?n)
   (O)
 where ?n > 2
 -> let ?d := floor(?n/2)
    AND(I[1..?d])
       (I.1)
    AND(I[?d+1..?n])
       (I.2)
    AND(I.1..2)
       (O);

OR(I)
  (O)
 -> let O := I;

OR(I^?n)
  (O)
 where ?n > 2
 -> let ?d := floor(?n/2)
    OR(I[1..?d])
      (I.1)
    OR(I[?d+1..?n])
      (I.2)
    OR(I.1..2)
      (O);

NAND(I)
    (O)
 -> let O := I';

NAND(I^?n)
    (O)
 where ?n > 2
 -> let ?d := floor(?n/2)
    NAND(I[1..?d])
        (I.1)
    NAND(I[?d+1..?n])
        (I.2)
    OR(I.1..2)
      (O);

NOR(I)
   (O)
 -> let O := I';
NOR(I^?n)
   (O)
 where ?n > 2
 -> let ?d := floor(?n/2)
    NOR(I[1..?d])
       (I.1)
    NOR(I[?d+1..?n])
       (I.2)
    AND(I.1..2)
       (O);
```

```
XOR(I)
   (O)
 -> let O := I;

XOR(I^?n)
   (O)
 where ?n > 2
 -> let ?d := floor(?n/2)
    XOR(I[1..?d])
       (I.1)
    XOR(I[?d+1..?n])
       (I.2)
    XOR(I.1..2)
       (O);

XNOR(I)
    (O)
 -> let O := I;

XNOR(I^?n)
    (O)
 where ?n > 2
 -> let ?d := floor(?n/2)
    XNOR(I[1..?d])
        (I.1)
    XNOR(I[?d+1..?n])
        (I.2)
    XNOR(I.1..2)
        (O);

/* Bitwise Logic Gates */

AND(I^?m:?n)
   (O^?n)
  where ?n > 1
  -> for ?i from 1 to ?n
       AND(I[1..?m:?i])
          (O[?i]);

OR(I^?m:?n)
  (O^?n)
  where ?n > 1
  -> for ?i from 1 to ?n
       OR(I[1..?m:?i])
         (O[?i]);

NAND(I^?m:?n)
    (O^?n)
  where ?n > 1
  -> for ?i from 1 to ?n
       NAND(I[1..?m:?i])
           (O[?i]);
```

```
NOR(I^?m:?n)
   (O^?n)
  where ?n > 1
  -> for ?i from 1 to ?n
       NOR(I[1..?m:?i])
          (O[?i]);

XOR(I^?m:?n)
   (O^?n)
  where ?n > 1
  -> for ?i from 1 to ?n
       XOR(I[1..?m:?i])
          (O[?i]);

XNOR(I^?m:?n)
    (O^?n)
  where ?n > 1
  -> for ?i from 1 to ?n
       XNOR(I[1..?m:?i])
           (O[?i]);
```

A.1.2 DECODE.SYN

The file DECODE.SYN defines n-bit binary and BCD decoders and encoders.

```
/* DECODE.SYN */

COMPONENT TYPES:

DECODE(I^?n)
      (O^?m)
  :style {BINARY|BCD} (default: BINARY)

ENCODE(I^?n)
      (O^?m)
  :style {BINARY|BCD} (default: BINARY)

DESIGN RULES:

/* ?N-TO-?M BINARY DECODER */

DECODE(I^?n)
      (O^?m)
  :style BINARY
  where ?m = 2^?n
  -> for ?i from 1 to ?m
       AND(X.?i.1..?n)
          (O[?i])
     for ?i from 1 to ?n
       {
         let ?s := 2^(?n-?i)
         for ?j from 1 to 2^?n by 2*?s
           {
             for ?k from ?j to ?j+?s-1
               let X.?k.?i := I[?n-?i+1]'
             for ?k from ?j+?s to ?j+2*?s-1
               let X.?k.?i := I[?n-?i+1]
           }
       };
```

```
/* ?M-TO-?N BINARY ENCODER (NONPRIORITY)

   Constructed of ?n ?m/2-bit OR gates, e.g., a 8-to-3 encoder uses
   three 4-bit OR gates.

   For each OR gate, the trick is to figure out which ?m/2 for the
   ?m inputs to attach to the inputs of the gate.  This is determined
   by the test

        mod(?i-1,2^?j) >= 2^(?j-1)

   If true, then I[?i] is an input to the ?j'th OR gate.

   The harder trick is figuring out which of the ?m/2 input pins I[?i]
   should be connected to.  The rather complicated equation

        (?i-2^(?j-1)*floor(1+?i/2^?j))

   computes this index.

   For example, in a 8-to-3 encoder, we first construct three OR gates:

        OR(A.1.1..4)    OR(A.2.1..4)    OR(A.3.1..4)
          (O[1])          (O[2])          (O[3])

   and end up with:

        OR(I[1] I[3] I[5] I[7])
          (O[1])
        OR(I[3] I[4] I[6] I[7])
          (O[2])
        OR(I[4] I[5] I[6] I[7])
          (O[3])

   To understand the index computation, consider it in parts:

        p = m/2
        r = m/2^j
        s = p/r
        g = floor(i/(m/r))
        o = (g*s)+s-1

        A.j.(i-o) := I[i]

   For each j, there will be p I[i], which are partitioned into r groups
   of s consecutive I[i], e.g., for 8-to-3, j=1 has 4 groups of 1;
   j=2 has 2 groups of 2; j=3 has 1 group of 4.  groups are labeled
   0 to s-1.  g computes the group of I[i].  o computes the offset
   from the group base, which is subtracted from i to give the index.
*/
```

```
ENCODE(I^?m)
      (O^?n)
  :style BINARY
  where ?n = log(?m,2)
  -> for ?i from 1 to ?n
       let O[?i] := OR(A.?i.1..?m/2)
     for ?i from 1 to ?m
       for ?j from 1 to ?n
         if mod(?i-1,2^?j) >= 2^(?j-1)
           let A.?j.(?i+1-2^(?j-1)*floor(1+?i/2^?j)) := I[?i];
```

A.1.3 MUX.SYN

The file MUX.SYN defines bitwise multiplexers (MUX), i.e., which select from m n-bit inputs that are directed to an n-bit output. The select port can be unary or binary encoded. The order in which inputs are mapped to select logic defaults to least significant to most significant, but the order can also be defined by the user.

```
/* MUX.SYN */

COMPONENT TYPES:

MUX(I^?m:?n S^?m)
   (O^?n)
  :ctrl S[] (:keys (1..?m) :style UNARY)

MUX(I^?m:?n S^?m)
   (O^?n)
  :ctrl S[] (:keys (1..?m) :style UNARY)
  :select ?idx
  where length(?idx) = ?m

MUX(I^?m:?n S^?s)
   (O^?n)
  :ctrl S[] (:keys (1..2^?s) :style BINARY)
  where ?m <= 2^?s

MUX(I^?m:?n S^?s)
   (O^?n)
  :ctrl S[] (:keys (1..2^?s) :style BINARY)
  :select ?idx
  where ?m <= 2^?s & ?m 1= length(?idx)

DESIGN RULES:

MUX(I^?n S^?s)
   (O)
  :ctrl S[]
  -> for ?i from 1 to ?n
       let A.?i := I[?i]*S[]:?i
     let O := OR(A.1..?n);

MUX(I^?m:?n S^?s)
   (O^?n)
  :ctrl S[]
  where ?n > 1
  -> for ?i from 1 to ?n
       MUX(I[1..?m:?i] S^?s)
          (O[?i]);

MUX(I^?n S^?s)
   (O)
  :ctrl S[]
  :select ?idx
  -> for ?i in ?idx as ?j from 1
       let A.?j := I[?i]*S[]:?j
     let O := OR(A.1..2^?s);
```

```
MUX(I^?m:?n S^?s)
   (O^?n)
  :ctrl S[]
  :select ?idx
  where ?n > 1
  -> for ?i from 1 to ?n
       MUX(I[1..?m:?i] S^?s)
          (O[?i])
         :select ?idx;
```

A.1.4 COMPARE.SYN

The file COMPARE.SYN defines an n-bit comparator (COMPARE) that can test as many as six relations: equal to (EQ), not equal to (NEQ), greater than (GT), less than (LT), greater than or equal to (GEQ), and less than or equal to (GEQ). This component type optionally accepts an carry-equal, carry-greater than, and carry-less than inputs. There is no operator select logic; all specified operations are computed in parallel and directed to distinct 1-bit outputs.

```
/* COMPARE.SYN */

COMPONENT TYPES:

COMPARE(A^?n B^?n [CLT] [CEQ] [CGT])
       (R^?r)
  :operations ?ops
  where ?ops in (EQ NEQ GT LT GEQ LEQ)
      & ?r = length(?ops)

DESIGN RULES:

COMPARE(A B [CEQ] [CGT] [CLT])
       (R^?r)
  :operations ?ops
  -> let CEQ := 1 (default)
        EQ  := CEQ*(A'*B+A*B')'
        LT  := CGT*A'*B
        GT  := CLT*A*B'
        NEQ := EQ'
        GEQ := LT'
        LEQ := GT'
    for ?i from ?r by -1 as ?op in ?ops
      if ?op = EQ
        let R[?i] := EQ
      else
      if ?op = NEQ
        let R[?i] := NEQ
      else
      if ?op = GT
        let R[?i] := GT
      else
      if ?op = LT
        let R[?i] := LT
      else
      if ?op = GEQ
        let R[?i] := GEQ
      else
      if ?op = LEQ
        let R[?i] := LEQ;
```

```
COMPARE(A^?n B^?n [CEQ] [CGT] [CLT])
       (R^?r)
  :operations ?ops
  where ?n > 1
  -> for ?i from 1 to ?n
       let EQ.?i := (A[?i]'*B[?i] + A[?i]*B[?i]')'
     for ?i from 1 to ?n-1
       let LT.?i := A[?i]'*B[?i]*EQ.?i+1..?n
           GT.?i := A[?i]*B[?i]'*EQ.?i+1..?n
     let LT.?n := A[?n]'*B[?n]
         GT.?n := A[?n]*B[?n]'
     let CEQ := 1 (default)
         EQ  := CEQ*EQ.1..?n
         GT  := CGT*EQ.1..?n + GT.1..?n
         LT  := CLT*EQ.1..?n + LT.1..?n
         NEQ := EQ'
         GEQ := LT'
         LEQ := GT'
     for ?i from ?r by -1 as ?op in ?ops
       if ?op = EQ
         let R[?i] := EQ
       else
       if ?op = NEQ
         let R[?i] := NEQ
       else
       if ?op = GT
         let R[?i] := GT
       else
       if ?op = LT
         let R[?i] := LT
       else
       if ?op = GEQ
         let R[?i] := GEQ
       else
       if ?op = LEQ
         let R[?i] := LEQ;
```

A.1.5 ADD.SYN

The file ADD.SYN defines an n-bit adder (ADD) with optional carry input and carry enable, as well as carry output and carry propagate and generate. There are two fundamental styles of n-bit adders: *ripple carry* and *carry look-ahead.* There are also two fundamental styles of 1-bit adders: *AND/OR implemented* and *NAND implemented*; the latter is typically faster in CMOS technologies. There is also an inverted style of adder used in combining the partial products of an array multiplier.

```
/* ADD.SYN */
COMPONENT TYPES:
/* Inverted adder for use in matrix multiplier */
ADD(XN^?n)
   (SN CN^floor(?n/2))
  :data XN[] SN
  :carry CN[]
  :style INVERTED
/* 1-bit adder with optional carry enable */
ADD(X Y [CI] [CE])
   (S [CO] [P G])
  :data X Y S
  :carry CI CO
  :cprop P
  :cgenr G
  :enable CE (:ports CI)
ADD(X Y [CI] [CE])
   (S [CO] [P G])
  :data X Y S
  :carry CI CO
  :cprop P
  :cgenr G
  :enable CE (:ports CI)
  :style {ANDOR|NAND}
/* ?n-bit adder with optional carry enable */
ADD(X^?n Y^?n [CI] [CE])
   (S^?n [CO] [P G])
  :data X[] Y[] S[]
  :carry CI CO
  :cprop P
  :cgenr G
  :enable CE (:ports CI)
  where ?n > 1
ADD(X^?n Y^?n [CI] [CE])
   (S^?n [CO] [P G])
  :data X[] Y[] S[]
  :carry CI CO
  :cprop P
  :cgenr G
  :enable CE (:ports CI)
  :style {RIPPLE|CLA}
  :levels {#'integerp|FULL}
```

```
DESIGN RULES:

/* generic ?n-bit adder; varies from a ripple carry adder with no
   embedded look ahead to a full CLA adder */

ADD(X^?n Y^?n [CI] [CE])
   (S^?n [CO])
  where ?n > 1
  varying ?l from 0 to ceiling(log(?n,4))
  -> ADD(X^?n Y^?n CI CE)
        (S^?n CO)
       :style RIPPLE
       :levels ?l;

/* when P and G are required, ripple carry.  try variations on CLA adder */

ADD(X^?n Y^?n [CI] [CE])
   (S^?n [CO] P G)
  where ?n > 1
  varying ?l from 1 to ceiling(log(?n,4))
  -> ADD(X^?n Y^?n CI CE)
        (S^?n CO P G)
       :style CLA
       :levels ?l;

/* ?n-bit ripple carry adder with no embedded look ahead */

ADD(X^?n Y^?n [CI] [CE])
   (S^?n [CO])
  :style RIPPLE
  :levels 0
  where ?n > 1
  -> for ?i from 1 to ?n
       ADD(X[?i] Y[?i] C.?i CE)
          (S[?i] C.?i+1)
     let C.1 := CI
         C.?n+1 := CO;
```

```
/* ?n-bit ripple carry adder with 1 or more levels of embedded look ahead */
ADD(X^?n Y^?n [CI] [CE])
   (S^?n [CO])
  :style RIPPLE
  :levels ?l|#'integerp
  where ?n > 1 & ?l > 0
  ->
     /* with ?l levels of look ahead and a 4-bit CLA...
           ?m is the width of each CLA adder
           ?r is the portion of ?n not covered by an ?m-bit adder */
     let ?m := 4^?l
         ?r := mod(?n,?m)
     if ?r = 0
       {
         /* ?n is completely covered by ?m-bit adders */
         for ?i from 1 to ?n by ?m
           ADD(X[?i..?i+?m-1] Y[?i..?i+?m-1] C.?i CE)
              (S[?i..?i+?m-1] C.?i+?m)
             :style CLA
             :levels ?l
         let C.1 := CI
             C.?n+1 := CO
       }
     else
     /* if some portion of ?n is not covered by an ?m-bit adder */
     if ?m < ?n
       {
         /* there's at least 1 ?m-bit adder covering ?n */
         for ?i from 1 to ?n-?r by ?m
           ADD(X[?i..?i+?m-1] Y[?i..?i+?m-1] C.?i CE)
              (S[?i..?i+?m-1] C.?i+?m)
             :style CLA
             :levels ?l
         /* a CLA adder covering the remainder of ?n is needed */
         ADD(X[?n-?r+1..?n] Y[?n-?r+1..?n] C.?n-?r+1 CE)
            (S[?n-?r+1..?n] CO)
           :style CLA
           :levels ?l-1
         let C.1 := CI
       }
     else
       ADD(X^?n Y^?n CI CE)
          (S^?n CO)
         :style CLA
         :levels ?l;
```

```
/* ?n-bit CLA adder (possible ripple to remainder) */

ADD(X^?n Y^?n [CI] [CE])
   (S^?n [CO] [P G])
  :style CLA
  :levels FULL
  where ?n > 1
  -> ADD(X^?n Y^?n CI CE)
        (S^?n CO P G)
       :style CLA
       :levels ceiling(log(?n,4));

/* ?n-bit CLA adder with 0 levels of look ahead is a ripple carry adder */

ADD(X^?n Y^?n [CI] [CE])
   (S^?n [CO])
  :style CLA
  :levels 0
  where ?n > 1
  -> ADD(X^?n Y^?n CI CE)
        (S^?n CO)
       :style RIPPLE
       :levels 0;

/* ?n-bit CLA adder with 1 level of look ahead (ripples CLA) */

ADD(X^?n Y^?n [CI] [CE])
   (S^?n [CO] [P G])
  :style CLA
  :levels 1
  where ?n > 1
  -> for ?i from 1 to ?n
       ADD(X[?i] Y[?i] C.?i CE)
          (S[?i] <> P.?i G.?i)
     CLA(P.1..?n G.1..?n C.1)
        (C.2..?n CO P G)
     let C.1 := CI;
```

```
/* ?n-bit CLA adder with 2 or more levels of look ahead */

ADD(X^?n Y^?n [CI] [CE])
   (S^?n [CO] [P G])
  :style CLA
  :levels ?l|#'integerp
  where ?n > 1 & ?l > 1
  ->
     /* with ?l levels of look ahead and a 4-bit CLA...
           ?m is the width of a level ?l CLA adder
           ?q is the width of a level ?l-1 CLA adder
           ?f is the number of ?l-1 CLA adders needed for ?n
           ?r is the remainder of ?n not covered by ?l-1 CLA adders */
     let ?m := 4^?l
         ?q := 4^(?l-1)
         ?f := floor(?n/?q)
         ?r := mod(?n,?q)
     if ?m < ?n
       {
         /* ?m does not completely cover ?n, so make a ripple adder with
            embedded carry */
         ADD(X^?n Y^?n CI CE)
            (S^?n CO P G)
           :style RIPPLE
           :levels ?l
       }
     else
     if ?r = 0
       {
         /* ?n is completely covered by ?n/?q (< 4) ?l-1 CLA adders */
         for ?i from 1 to ?n by ?q as ?j from 1
           ADD(X[?i..?i+?q-1] Y[?i..?i+?q-1] C.?j CE)
              (S[?i..?i+?q-1] <> P.?j G.?j)
             :style CLA
             :levels ?l-1
         CLA(P.1..?n/?q G.1..?n/?q C.1)
            (C.2..?n/?q CO P G)
         let C.1 := CI
       }
     else
```

```
      if ?f > 0
        {
          /* ?n is covered by ?n/?q ?l-1 CLA adders plus an ?r-bit adder */
          for ?i from 1 to ?n-?r by ?q as ?j from 1
            ADD(X[?i..?i+?q-1] Y[?i..?i+?q-1] C.?j CE)
               (S[?i..?i+?q-1] <> P.?j G.?j)
              :style CLA
              :levels ?l-1
          ADD(X[?n-?r+1..?n] Y[?n-?r+1..?n] C.?f+1 CE)
             (S[?n-?r+1..?n] <> P.?f+1 G.?f+1)
            :style CLA
            :levels ?l-1
          CLA(P.1..?f+1 G.1..?f+1 C.1)
             (C.2..?f+1 CO P G)
          let C.1 := CI
        }
      else
        {
          /* otherwise, ?n is two small for a CLA adder, so just ripple */
          ADD(X^?n Y^?n CI CE)
             (S^?n CO P G)
            :style RIPPLE
            :levels 0
        };

/* 1-bit RIPPLE/CLA adder is just a 1-bit adder */

ADD(X Y [CI] [CE])
   (S [CO] [P G])
  :style {RIPPLE|CLA}
  :levels {#'integerp|FULL}
  -> ADD(X Y CI CE)
        (S CO P G);

/* 1-bit adder with carry enable */

ADD(X Y CI CE)
   (S [CO] [P G])
  -> ADD(X Y C)
        (S CO P G)
     let C := CI*CE;

ADD(X Y [CI])
   (S [CO] [P G])
  varying ?sty in (ANDOR NAND)
  -> ADD(X Y CI)
        (S CO P G)
       :style ?sty;
```

```
/* 1-bit adder AND/OR encoding CO */

ADD(X Y [CI])
   (S [CO] [P G])
  :style ANDOR
  -> let P  := X(+)Y
        S  := CI(+)P
        G  := X*Y
        CO := (CI*P)+G

/* 1-bit adder NAND encoding CO */

ADD(X Y [CI])
   (S [CO] [P G])
  :style NAND
  -> let P  := X(+)Y
        S  := CI(+)P
        GN := !(X*Y)
        CO := !(!(CI*P)*GN)
        G  := !GN

/* inverted adders */

ADD(XN^2)
   (SN CN)
  :style INVERTED
  -> let SN := XN[1](*)XN[2]
        CN := XN[1]+XN[2]

ADD(XN^3)
   (SN CN)
  :style INVERTED
  -> let PN := XN[1](*)XN[2]
        SN := XN[3](*)PN
        CN := !(!(XN[1]+XN[2])+!(PN+XN[3]))

ADD(XN^?n)
   (SN CN^floor(?n/2))
  :style INVERTED
  where ?n > 3
  -> ADD(XN[1..3])
       (A CN[1])
      :style INVERTED
     ADD(<A XN[4..?n]>)
       (SN CN[2..floor(?n/2)])
      :style INVERTED
```

A.1.6 CLA.SYN

The file CLA.SYN defines n-bit carry look-ahead generators (CLA) and carry propagate and generate generators (PG). For $n > 4$, CLAs are decomposed into a series of $\frac{n}{4}$-bit CLAs. For $n \leq 4$, CLAs and PGs there are two styles: *AND/OR implemented* and *NAND implemented*; the latter is typically faster in CMOS technologies.

```
/* GATES.SYN */

COMPONENT TYPES:

/* carry look-ahead generator */

CLA(P^?n G^?n CI)
   (C^?n-1 [CO] [P G])
  :cprop P[] P
  :cgenr G[] G
  :carry CI CO C[]

CLA(P^?n G^?n CI)
   (C^?n-1 [CO] [P G])
  :cprop P[] P
  :cgenr G[] G
  :carry CI CO C[]
  :style {ANDOR|NAND}

/* P and G generator */

PG(P^?n G^?n)
  (P G)
  :cprop P[] P
  :cgenr G[] G
  :style {ANDOR|NAND}

DESIGN RULES:

CLA(P.1..?n G.1..?n [CI])
   (C.2..?n [CO] [P G])
  varying ?sty in (ANDOR NAND)
  -> CLA(P.1..?n G.1..?n CI)
        (C.2..?n CO P G)
       :style ?sty;

/* CLA with optional carry output */

CLA(P.1..?n G.1..?n CI)
   (C.2..?n [CO])
  :style ?sty
  where ?n > 4
  -> CLA(P.1..4 G.1..4 CI)
        (C.2..4 C.5)
       :style ?sty
     CLA(P.5..?n G.5..?n C.5)
        (C.6..?n CO)
       :style ?sty;
```

```
/* CLA with optional carry output and P and G */

CLA(P.1..?n G.1..?n CI)
   (C.2..?n [CO] P G)
  :style ?sty
  -> if ?n <= 4
       {
         CLA(P.1..?n G.1..?n CI)
            (C.2..?n CO)
           :style ?sty
         PG(P.1..?n G.1..?n)
           (P G)
           :style ?sty
       }
     else
       {
         CLA(P.1..4 G.1..4 CI)
            (C.2..4 C.5)
           :style ?sty
         CLA(P.5..?n G.5..?n C.5)
            (C.6..?n CO P G)
           :style ?sty
       };

/* CLA with optional carry output; AND/OR implementation */

CLA(P.1..?n G.1..?n CI)
   (C.2..?n [CO])
  :style ANDOR
  where ?n <= 4
  -> let C.2 := C.1*P.1 + G.1
     for ?i from 2 to ?n
       {
         let A.?i.1 := C.1*P.1..?i
         for ?j from ?i-1 to 1 by -1
           let A.?i.?j+1 := G.?j*P.?j+1..?i
         let C.?i+1 := G.?i + A.?i.1..?i
j      }
     let C.1 := CI
         C.?n+1 := CO;
```

```
/* CLA with optional carry output; NAND implementation */

CLA(P.1..?n G.1..?n CI)
   (C.2..?n [CO])
  :style NAND
  where ?n <= 4
  -> let C.2 := !(!G.1*!(C.1*P.1))
     for ?i from 2 to ?n
       {
         let A.?i.1 := !(C.1*P.1..?i)
         for ?j from ?i-1 to 1 by -1
           let A.?i.?j+1 := !(G.?j*P.?j+1..?i)
         let C.?i+1 := !(!G.?i*A.?i.1..?i)
       }
     let C.1 := CI
         C.?n+1 := CO;

/* P and G generator using AND/OR implementation */

PG(P.1..?n G.1..?n)
  (P G)
  :style ANDOR
  -> for ?i from 1 to ?n-1
       let A.?i := G.?i*P.?i+1..?n
     let G  := A.1..?n-1 + G.?n
         P  := *(P.1..?n);

/* P and G generator using NAND implementation */

PG(P.1..?n G.1..?n)
  (P G)
  :style NAND
  -> for ?i from 1 to ?n-1
       let A.?i := !(G.?i*P.?i+1..?n)
     let G  := !(!G.?n*A.1..?n-1)
         P  := *(P.1..?n);
```

A.1.7 MULT.SYN

The file MULT.SYN defines an n-by-m multiplier (MULT), i.e., where one input is n-bits and the other is m-bits; the single output has width $n + m$. There are two styles of multipliers: *matrix* (or *array*) and *Wallace tree* (or *tree*). This SYN file also defines a type of 4-input carry-save (CSA) adder for use in its definition of a tree multiplier.

```
/* MULT.SYN */

COMPONENT TYPES:

MULT(X^?n Y^?m)
    (P^?n+?m)

MULT(X^?n Y^?m)
    (P^?n+?m)
  :style PARALLEL
  where ?n < 4 & ?m < 4

MULT(X^?n Y^?m)
    (P^?n+?m)
  :style {MATRIX|TREE}

MULTROW(X^?n Y)
       (PN^?n)

/* CSA plain adder */

ADD(PP1^?r PP2^?n PP3^?m PP4^?s)
   (S^max(?m,?n) C^max(?m,?n))
  :style CSA
```

```
DESIGN RULES:

MULT(X^?n Y^?m)
    (P^?n+?m)
  varying ?sty in (MATRIX TREE)
  -> MULT(X^?n Y^?m)
        (P^?n+?m)
      :style ?sty;

/* MATRIX MULTIPLIER USING INVERTED ADDERS */

MULT(X^?n Y^?m)
    (P^?n+?m)
  :style MATRIX
  where ?n >= ?m & ?n > 1 & ?n < 49
  -> for ?i from 1 to ?m
       MULTROW(X[1..?n] Y[?i])
              (P.?i..?i+?n-1.?i)
     let P[1] := !P.1.1
     ADD(P.2.1..2)
        (PN[2] C.3.1)
       :style INVERTED
     for ?i from 3 to ?m
       ADD(<P.?i.1..?i C.?i.1..?i-2>)
          (PN[?i] C.?i+1.1..?i-1)
         :style INVERTED
     for ?i from ?m+1 to ?n
       ADD(<P.?i.1..?m C.?i.1..?m-1>)
          (PN[?i] C.?i+1.1..?m-1)
         :style INVERTED
     ADD(<P.?n+1.2..?m C.?n+1.1..?m-1>)
        (PN[?n+1] C.?n+2.1..?m-1)
       :style INVERTED
     for ?i from ?n+2 to ?n+?m-1 as ?j from ?m-1 by -1
       ADD(<P.?i.?i-?n+1..?m C.?i.1..?j>)
          (PN[?i] C.?i+1.1..?j-1)
         :style INVERTED
     INV(<PN[2..?n+?m-1] C.?n+?m.1>)
        (P[2..?n+?m]);

MULTROW(X^?n Y)
       (PN^?n)
  -> for ?i from 1 to ?n
       let PN[?i] := !(X[?i]*Y);
```

```
/* HAND-CODED PARALLEL MULTIPLIERS (BASE CASE FOR TREE MULTIPLIERS) */

MULT(X^2 Y^2)
    (Z^4)
  :style PARALLEL
  -> X[2] X[1] Y[2] Y[1] | Z[4] Z[3] Z[2] Z[1]
     ---------------------------------------
     0    0    X    X    | 0    0    0    0
     X    X    0    0    | 0    0    0    0
     0    1    X    X    | 0    0    Y[2] Y[1]
     1    0    X    X    | 0    Y[2] Y[1] 0
     X    X    0    1    | 0    0    X[2] X[1]
     X    X    1    0    | 0    X[2] X[1] 0
     1    1    1    1    | 1    0    0    1

MULT(X^2 Y^3)
    (Z^5)
  :style PARALLEL
  -> MULT(Y^3 X^2)
         (Z^5)
      :style PARALLEL;

MULT(X^3 Y^2)
    (Z^5)
  :style PARALLEL
  -> X[3] X[2] X[1] Y[2] Y[1] | Z[5] Z[4] Z[3] Z[2] Z[1]
     -------------------------------------------------
     0    0    0    X    X    | 0    0    0    0    0
     X    X    X    0    0    | 0    0    0    0    0
     0    0    1    X    X    | 0    0    0    Y[2] Y[1]
     0    1    0    X    X    | 0    0    Y[2] Y[1] 0
     1    0    0    X    X    | 0    Y[2] Y[1] 0    0
     X    X    X    0    1    | 0    0    X[3] X[2] X[1]
     X    X    X    1    0    | 0    X[3] X[2] X[1] 0
     0    1    1    1    1    | 0    1    0    0    1
     1    0    1    1    1    | 0    1    1    1    1
     1    1    0    1    1    | 1    0    0    1    0
     1    1    1    1    1    | 1    0    1    0    1
```

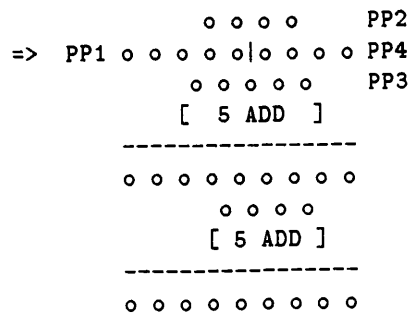
```
MULT(X^3 Y^3)
    (Z^6)
  :style PARALLEL
  -> X[3] X[2] X[1] Y[3] Y[2] Y[1] | Z[6] Z[5] Z[4] Z[3] Z[2] Z[1]
     ------------------------------------------------------------
     0    0    0    X    X    X    | 0    0    0    0    0    0
     X    X    X    0    0    0    | 0    0    0    0    0    0
     0    0    1    X    X    X    | 0    0    0    Y[3] Y[2] Y[1]
     0    1    0    X    X    X    | 0    0    Y[3] Y[2] Y[1] 0
     1    0    0    X    X    X    | 0    Y[3] Y[2] Y[1] 0    0
     X    X    X    0    0    1    | 0    0    0    X[3] X[2] X[1]
     X    X    X    0    1    0    | 0    0    X[3] X[2] X[1] 0
     X    X    X    1    0    0    | 0    X[3] X[2] X[1] 0    0
     0    1    1    0    1    1    | 0    0    1    0    0    1
     0    1    1    1    0    1    | 0    0    1    1    1    1
     0    1    1    1    1    0    | 0    1    0    0    1    0
     0    1    1    1    1    1    | 0    1    0    1    0    1
     1    0    1    0    1    1    | 0    0    1    1    1    1
     1    0    1    1    0    1    | 0    1    1    0    0    1
     1    0    1    1    1    0    | 0    1    1    1    1    0
     1    0    1    1    1    1    | 1    0    0    0    1    1
     1    1    0    0    1    1    | 0    1    0    0    1    0
     1    1    0    1    0    1    | 0    1    1    1    1    0
     1    1    0    1    1    0    | 1    0    0    1    0    0
     1    1    0    1    1    1    | 1    0    1    0    1    0
     1    1    1    0    1    1    | 0    1    0    1    0    1
     1    1    1    1    0    1    | 1    0    0    0    1    1
     1    1    1    1    1    0    | 1    0    1    0    1    0
     1    1    1    1    1    1    | 1    1    0    0    0    1
```

```
/* TREE MULTIPLIERS  */

/* switch to base case */

MULT(X^?n Y^?m)
    (Z^?n+?m)
  :style TREE
  where ?n >= ?m & (?n = 2 | ?n = 3)
  -> MULT(X^?n Y^?m)
         (Z^?n+?m)
      :style PARALLEL;

MULT(X^4 Y^3)
    (Z^7)
  :style TREE
  -> MULT(X^4 Y^3)
         (Z^7)
      :style MATRIX;

/* switch arguments so that n > m */

MULT(X^?n Y^?m)
    (Z^?n+?m)
  :style TREE
  where ?m > ?n
  -> MULT(Y^?m X^?n)
         (Z^?n+?m)
      :style TREE;
```

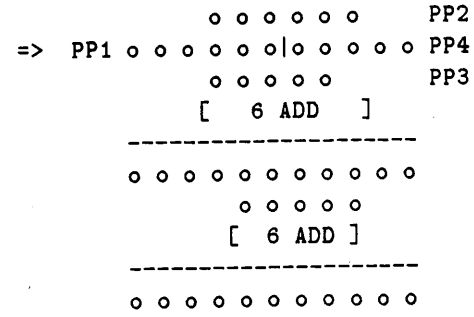
```
/* N-TO-2N MULTIPLIER, WHERE N IS EVEN

   For example, 4-to-8
     P1 o o   o o
          o o   o o P2                  o o o o     PP2
                            =>  PP1 o o o o|o o o o PP4
        P3 o o   o o                    o o o o     PP3
             o o   o o P4             [ 4 ADD ]
                                     ---------------
                                     o o o o o o o o
                                          o o o o
                                       [ 4 ADD ]
                                     ---------------
                                     o o o o o o o o
*/
MULT(X^?n Y^?n)
    (P^?n*2)
  :style TREE
  where ?n >= 4 & evenp(?n)
  varying ?sty in (MATRIX TREE)
  -> let ?r := ?n/2
         ?s := ?r+1
     MULT(X[1..?r] Y[1..?r])
         (PP1^?n)
       :style ?sty
     MULT(X[?s..?n] Y[1..?r])
         (PP2^?n)
       :style ?sty
     MULT(X[1..?r] Y[?s..?n])
         (PP3^?n)
       :style ?sty
     MULT(X[?s..?n] Y[?s..?n])
         (PP4^?n)
       :style ?sty
     for ?i from 1 to ?r
       let PP1[?i] := P[?i]
     ADD(PP1[?s..?n] PP2[1..?n] PP3[1..?n] PP4[1..?r])
        (<P[?s] A[1..?n-1]> B[1..?n])
       :style CSA
     let PP4[?s] := A[?n]
     ADD(A[1..?n] B[1..?n])
        (P[?s+1..?s+?n] CO)
     let P[?s+?n+1] := PP4[?s+1] + CO
     for ?i from ?s+?n+2 to 2*?n as ?j from ?s+2
       let PP4[?j] := P[?i];
```

```
/* N-TO-2N MULTIPLIER, WHERE N IS ODD

   For example, 5-TO-10
        P1 o o o    o o
             o o o    o o                                  o o o o o     PP2
               o o o    o o P2       =>  PP1 o o o o o o |o o o o PP4
                                             o o o o o     PP3
               P3 o o o    o o                 [  5 ADD  ]
                    o o o    o o P4          ------------------
                                            o o o o o o o o o o
                                                  o o o o o
                                                 [5-bit ADD] +
                                            ------------------
                                            o o o o o o o o o o

*/
MULT(X^?n Y^?n)
    (P^?n*2)
  :style TREE
  where ?n >= 4 & oddp(?n)
  varying ?sty in (MATRIX TREE)
  -> let ?r := ceiling(?n/2)
         ?s := ?r+1
     MULT(X[1..?r] Y[1..?r])
         (PP1^?r*2)
       :style ?sty
     MULT(X[?s..?n] Y[1..?r])
         (PP2^?r*2-1)
       :style ?sty
     MULT(X[1..?r] Y[?s..?n])
         (PP3^?r*2-1)
       :style ?sty
     MULT(X[?s..?n] Y[?s..?n])
         (PP4^(?r-1)*2)
       :style ?sty
     for ?i from 1 to ?r
       let PP1[?i] := P[?i]
     ADD(PP1[?s..?n] PP2[1..?n] PP3[1..?n] PP4[1..?r-1])
        (<P[?s] A[1..?n-1]> B[1..?n])
       :style CSA
     let PP4[?r] := A[?n]
     ADD(A[1..?n] B[1..?n])
        (P[?s+1..?s+?n] CO)
     let P[?s+?n+1] := PP4[?s] + CO
     for ?i from ?s+?n+2 to 2*?n as ?j from ?s+1
       let PP4[?j] := P[?i];
```

```
/* N-BY-M MULTIPLIER, WHERE N > M

   For example, 5-by-4

        P1 o o o    o o
             o o o    o o P2

                P3 o o o    o o
                     o o o    o o P4
```




```
   For example, 6-by-5

        P1 o o o    o o o
             o o o    o o o
               o o o    o o o P2

                P3 o o o    o o o
                     o o o    o o o P4
```




```
*/
```

```
MULT(X^?n Y^?m)
    (P^?n+?m)
  :style TREE
  where ?n > 4 & ?n = ?m+1
  varying ?sty in (MATRIX TREE)
  -> let ?r := ceiling(?n/2)
        ?s := ?r+1
        ?t := ceiling(?m/2)
        ?u := ?t+1
     MULT(X[1..?r] Y[1..?t])
        (PP1^?r+?t)
      :style ?sty
     MULT(X[?s..?n] Y[1..?t])
        (PP2^?n-?r+?t)
      :style ?sty
     MULT(X[1..?r] Y[?u..?m])
        (PP3^?r+?m-?t)
      :style ?sty
     MULT(X[?s..?n] Y[?u..?m])
        (PP4^(?n-?r)+(?m-?t))
      :style ?sty
     for ?i from 1 to ?t
       let PP1[?i] := P[?i]
     let ?z := ?n-?r
     ADD(PP1[?z+1..?r+?t] PP2[1..?z+?t] PP3[1..?r+?m-?t] PP4[1..?z+(?m-?t)])
        (<P[?u] A[1..?n-1]> B[1..?n])
       :style CSA
     let PP4[?u] := A[?n]
     ADD(A[1..?n] B[1..?n])
        (P[?u+1..?u+?n] CO)
     let P[?u+?n+1] := PP4[?u+1] + CO
     for ?i from ?u+?n+2 to ?n+?m as ?j from ?u+2
       let PP4[?j] := P[?i];
```

```
/* CSA adders for tree multipliers */

ADD(PP1^?r PP2^?n PP3^?n PP4^?s)
   (S^?n C^?n)
  :style CSA
  where ?n = ?r+?s
  -> for ?i from 1 to ?n
       ADD(PP2[?i] P.?i PP3[?i])
          (S[?i] C[?i])
     for ?i from 1 to ?r
       let P.?i := PP1[?i]
     for ?i from ?r+1 to ?n as ?j from 1
       let P.?i := PP4[?j];

ADD(PP1^?r PP2^?n PP3^?m PP4^?s)
   (S^?n C^?n)
  :style CSA
  where ?n > ?m
  -> for ?i from 1 to ?n-1
       ADD(PP2[?i] P.?i PP3[?i])
          (S[?i] C[?i])
     ADD(PP2[?n] P.?n)
        (S[?n] C[?n])
     for ?i from 1 to ?r
       let P.?i := PP1[?i]
     for ?i from ?r+1 to ?n as ?j from 1
       let P.?i := PP4[?j];

ADD(PP1^?r PP2^?n PP3^?m PP4^?s)
   (S^?m C^?m)
  :style CSA
  where ?n < ?m
  -> ADD(P.1 PP3[1])
        (S[1] C[1])
     for ?i from 2 to ?m
       ADD(PP2[?i-1] P.?i PP3[?i])
          (S[?i] C[?i])
     for ?i from 1 to ?r
       let P.?i := PP1[?i]
     for ?i from ?r+1 to ?m as ?j from 1
       let P.?i := PP4[?j];
```

A.1.8 ALU.SYN

The file ALU.SYN defines an n-bit arithmetic logic unit (ALU) with optional carry input, carry output, carry propagate and generate, and comparison output. The ALU can compute as many as four arithmetic operations: ADD, SUB, INC (increment by one), DEC (decrement by one); eight comparison operations: EQ, NEQ, GT, LT, GEQ, GEQ, ZEROP (equal to zero), and ONEP (equal to one); and 16 logic operations: AND, OR, NAND, NOR, XOR, XNOR, LID and RID (left or right input), LNOT and RNOT (left or right input inverted), LINHI and RINHI (left inhibits right or vice versa), LIMPL and RIMPL (left implies right or vice versa).

```
/* ALU.SYN */

COMPONENT TYPES:

ALU(A^?n B^?n [CI] S^?s)
   (F^?n [CO] [R] [V])
  :data A[] B[] F[] R V
  :ctrl S[] (:keys ?ops)
  :carry CI CO
  :operations ?ops
  where subsetp(?ops, (ADD SUB INC DEC EQ NEQ
                       LT LEQ GT GEQ ZEROP ONEP
                       ZERO ONE AND NAND OR NOR
                       XOR XNOR LID RID LNOT RNOT
                       LINHI RINHI LIMPL RIMPL))

AU(A^?n B^?n [CI] S^?s)
  (F^?n [CO] [R] [V])
  :data A[] B[] F[] R V
  :ctrl S[] (:keys ?ops)
  :carry CI CO
  :operations ?ops (optional)
  where subsetp(?ops, (ADD SUB INC DEC EQ NEQ
                       LT LEQ GT GEQ ZEROP ONEP))

AU(A^?n B^?n [CI] S^?s)
  (F^?n [CO] [R] [V])
  :data A[] B[] F[] R V
  :ctrl S[] (:keys ?keys)
  :carry CI CO
  :operations ?ops (default: ?keys in (ADD SUB INC DEC EQ NEQ
                                       LT LEQ GT GEQ ZEROP ONEP))
  where subsetp(?ops, ?keys)
      & subsetp(?ops, (ADD SUB INC DEC EQ NEQ
                       LT LEQ GT GEQ ZEROP ONEP))

FG(A^?n B^?n S^?s)
  (F^?n)
  :data A[] B[] F[]
  :ctrl S[] (:keys ?ops)
  :operations ?ops (optional)
  where subsetp(?ops, (ZERO ONE AND NAND OR NOR
                       XOR XNOR LID RID LNOT RNOT
                       LINHI RINHI LIMPL RIMPL))
```

```
FG(A^?n B^?n S^?s)
  (F^?n)
  :data A[] B[] F[]
  :ctrl S[] (:keys ?keys)
  :operations ?ops (default: ?keys in (ZERO ONE AND NAND OR NOR
                                       XOR XNOR LID RID LNOT RNOT
                                       LINHI RINHI LIMPL RIMPL))
  where subsetp(?ops, ?keys)
      & subsetp(?ops, (ZERO ONE AND NAND OR NOR
                       XOR XNOR LID RID LNOT RNOT
                       LINHI RINHI LIMPL RIMPL))

CC.XY(A^?n B^?n S^?s)
     (X^?n Y^?n)
  :data A[] B[] X[] Y[]
  :ctrl S[] (:keys ?keys)
  :select ?ops (default: ?keys)
  where subsetp(?ops, ?keys)

CC.CI([C1] S^?s)
     (C2 [CE])
  :data C1 C2 CE
  :ctrl S[] (:keys ?keys)
  :select ?ops (default: ?keys)
  where subsetp(?ops, ?keys)

CC.AL([A^?n B^?n] [C1] S^?s)
     ([X^?n Y^?n] [C2 [CE]])
  :data A[] B[] X[] Y[] C1 C2 CE
  :ctrl S[] (:keys ?keys)
  :select ?ops (default: ?keys)
  where subsetp(?ops, ?keys)

CC.EQ(F^?n CO S^?s)
     (R)
  :data F[] CO R
  :ctrl S[] (:keys ?keys)
  :select ?ops (default: ?keys in (EQ NEQ LT LEQ GT GEQ ZEROP ONEP))
  where subsetp(?ops, ?keys)
      & subsetp(?ops, (EQ NEQ LT LEQ GT GEQ ZEROP ONEP))

CC.EQ(F^?n [CO] S^?s)
     (R)
  :data F[] CO R
  :ctrl S[] (:keys ?keys)
  :select ?ops (default: ?keys in (EQ NEQ ZEROP ONEP))
  where subsetp(?ops, ?keys)
      & subsetp(?ops, (EQ NEQ ZEROP ONEP))
```

```
CC.EQ([F^?n] CO S^?s)
     (R)
  :data F[] CO R
  :ctrl S[] (:keys ?keys)
  :select ?ops (default: ?keys in (LT LEQ GT GEQ))
  where subsetp(?ops, ?keys)
      & subsetp(?ops, (LT LEQ GT GEQ))

CC.MX(S^?s)
     (Z)
  :data Z
  :ctrl S[] (:keys ?keys)
  :select ?ops
  where subsetp(?ops, ?keys)

DESIGN RULES:

ALU(A^?n B^?n [CI] S^?s)
   (F^?n [CO] [R])
  :operations ?ops
  where subsetp(?ops, (ADD SUB INC DEC))
  -> AU(A^?n B^?n CI S^?s)
       (F^?n CO)
       :operations ?ops
     let R := GND;

ALU(A^?n B^?n [CI] S^?s)
   (F^?n [CO] R)
  :operations ?ops
  where subsetp(?ops, (ADD SUB INC DEC EQ NEQ
                       LT LEQ GT GEQ ZEROP ONEP))
     & !subsetp(?ops, (ADD SUB INC DEC))
  -> AU(A^?n B^?n CI S^?s)
       (F^?n CO R)
       :operations ?ops;

ALU(A^?n B^?n [CI] S^?s)
   (F^?n [CO] [R])
  :operations ?ops
  where subsetp(?ops, (ZERO ONE AND NAND OR NOR
                       XOR XNOR LID RID LNOT RNOT
                       LINHI RINHI LIMPL RIMPL))
  -> FG(A^?n B^?n S^?s)
       (F^?n)
       :operations ?ops
     let CI := GND
         CO := GND
         R  := GND;
```

```
ALU(A^?n B^?n [CI.1] S^?s)
   (F^?n [CO] [R])
  :operations ?ops
  where !(EQ NEQ LT LEQ GT GEQ ZEROP ONEP) in ?ops
      & !subsetp(?ops, (ADD SUB INC DEC))
      & !subsetp(?ops, (ZERO ONE AND NAND OR NOR
                        XOR XNOR LID RID LNOT RNOT
                        LINHI RINHI LIMPL RIMPL))
  -> CC.XY(A^?n B^?n S^?s)
         (X^?n Y^?n)
       :select ?ops
     CC.CI(CI.1 S^?s)
         (CI.2 CE)
       :select ?ops
     ADD(X^?n Y^?n CI.2 CE)
        (F^?n CO)
     let R := GND;

ALU(A^?n B^?n [CI.1] S^?s)
   (F^?n [C] R)
  :operations ?ops
  where ?cmps := (EQ NEQ LT LEQ GT GEQ ZEROP ONEP) in ?ops
      & !subsetp(?ops, (ADD SUB INC DEC EQ NEQ
                        LT LEQ GT GEQ ZEROP ONEP))
  -> CC.XY(A^?n B^?n S^?s)
         (X^?n Y^?n)
       :select ?ops
     CC.CI(CI.1 S^?s)
         (CI.2 CE)
       :select ?ops
     ADD(X^?n Y^?n CI.2 CE)
        (F^?n CO)
     CC.EQ(F^?n CO S^?s)
         (R)
       :select ?cmps
     let C := CO;
```

```
ALU(A^?n B^?n [CI] S^?s)
   (F^?n [CO] [R])
  :operations ?ops
  where ?opsA := (ADD SUB INC DEC EQ NEQ
                  LT LEQ GT GEQ ZEROP ONEP) in ?ops
      & ?opsL := (ZERO ONE AND NAND OR NOR
                  XOR XNOR LID RID LNOT RNOT
                  LINHI RINHI LIMPL RIMPL) in ?ops
  -> AU(A^?n B^?n CI S^?s)
       (FA^?n CO R)
      :operations ?opsA
     FG(A^?n B^?n S^?s)
       (FL^?n)
      :operations ?opsL
     CC.MX(S^?s)
          (Z)
      :select ?opsA
     MUX(<FL^?n FA^?n> Z)
        (F^?n);

AU(A^?n B^?n [CI.1] S^?s)
  (F^?n [CO] [R])
  :operations ?ops
  where subsetp(?ops, (ADD SUB INC DEC))
  -> CC.XY(A^?n B^?n S^?s)
          (X^?n Y^?n)
       :select ?ops
     CC.CI(CI.1 S^?s)
          (CI.2)
       :select ?ops
     ADD(X^?n Y^?n CI.2)
        (F^?n CO)
     let R := GND;

AU(A^?n B^?n [CI.1] S^?s)
  (F^?n [C] R)
  :operations ?ops
  where ?cmps := (EQ NEQ LT LEQ GT GEQ ZEROP ONEP) in ?ops
        subsetp(?cmps, (LT LEQ GT GEQ))
  -> CC.XY(A^?n B^?n S^?s)
          (X^?n Y^?n)
       :select ?ops
     CC.CI(CI.1 S^?s)
          (CI.2)
       :select ?ops
     ADD(X^?n Y^?n CI.2)
        (F^?n CO)
     CC.EQ(<> CO S^?s)
          (R)
       :select ?cmps
     let C := CO;
```

```
AU(A^?n B^?n [CI.1] S^?s)
  (F^?n [CO] R)
  :operations ?ops
  where ?cmps := (EQ NEQ LT LEQ GT GEQ ZEROP ONEP) in ?ops
        subsetp(?cmps, (EQ NEQ ZEROP ONEP))
  -> CC.XY(A^?n B^?n S^?s)
           (X^?n Y^?n)
         :select ?ops
     CC.CI(CI.1 S^?s)
           (CI.2)
         :select ?ops
     ADD(X^?n Y^?n CI.2)
        (F^?n CO)
     CC.EQ(F^?n <> S^?s)
           (R)
       :select ?cmps;

AU(A^?n B^?n [CI.1] S^?s)
  (F^?n [C] R)
  :operations ?ops
  where ?cmps := (EQ NEQ LT LEQ GT GEQ ZEROP ONEP) in ?ops
        !subsetp(?cmps, (LT LEQ GT GEQ))
      & !subsetp(?cmps, (EQ NEQ ZEROP ONEP))
  -> CC.XY(A^?n B^?n S^?s)
           (X^?n Y^?n)
         :select ?ops
     CC.CI(CI.1 S^?s)
           (CI.2)
         :select ?ops
     ADD(X^?n Y^?n CI.2)
        (F^?n CO)
     CC.EQ(F^?n CO S^?s)
           (R)
         :select ?cmps
     let C := CO;

FG(A^?n B^?n S^?s)
  (F^?n)
  :operations ?ops
  where ?n > 1
  -> for ?i from 1 to ?n
       FG(A[?i] B[?i] S^?s)
         (F[?i])
        :operations ?ops;
```

```
FG(A B S^?s)
  (F)
  :operations ?ops
  ->            S[]:?ops | S.1 S.2 S.3 S.4
                ------------------------
                  ZERO   |  0   0   0   0
                  ONE    |  1   1   1   1
                  AND    |  1   0   0   0
                  NAND   |  0   1   1   1
                  OR     |  1   1   1   0
                  NOR    |  0   0   0   1
                  XOR    |  0   1   1   0
                  XNOR   |  1   0   0   1
                  LID    |  1   1   0   0
                  RID    |  1   0   1   0
                  LNOT   |  0   0   1   1
                  RNOT   |  0   1   0   1
                  LINHI  |  0   0   1   0
                  RINHI  |  0   1   0   0
                  LIMPL  |  1   1   0   1
                  RIMPL  |  1   0   1   1
     let F := S.1*A*B + S.2*A*B' + S.3*A'*B + S.4*A'*B'

CC.XY(A^?n B^?n S^?s)
     (X^?n Y^?n)
  :select ?ops
  -> for ?i from 1 to ?n
       CC.AL(A[?i] B[?i] <> S^?s)
            (X[?i] Y[?i])
         :select ?ops;

CC.CI([CI.1] S^?s)
     (CI.2 [CE])
  :select ?ops
  -> CC.AL(<> <> CI.1 S^?s)
           (<> <> CI.2 CE)
       :select ?ops;
```

```
CC.AL([A B] [CI] S^?s)
     ([X Y] [CO [CE]])
  :select ?ops
  -> S[]:?ops |   X    Y   CO   CE
     -------------------------
       ADD    |   A    B   CI   1
       SUB    |   A    B'  CI'  1
       INC    |   A    0   CI'  1
       DEC    |   A    1   CI   1
     -------------------------
       EQ     |   A    B'  CI'  1
       NEQ    |   A    B'  CI'  1
       LT     |   A    B'  CI'  1
       LEQ    |   A    B'  CI   1
       GT     |   A    B'  CI   1
       GEQ    |   A    B'  CI'  1
       ZEROP  |   A    0   CI   1
       ONEP   |   A    1   CI'  1
     -------------------------
       ZERO   |   0    0   --   0
       ONE    |   1    0   --   0
       AND    |  A+B'  B'  --   0
       NAND   | A'+B'  0   --   0
       OR     |  A+B   0   --   0
       NOR    | A'+B   B   --   0
       XOR    |   A    B   --   0
       XNOR   |   A    B'  --   0
       LID    |   A    0   --   0
       RID    |   0    B   --   0
       LNOT   |   A'   0   --   0
       RNOT   |   0    B'  --   0
       LINHI  | A'+B'  B'  --   0
       RINHI  |  A+B   B   --   0
       LIMPL  | A'+B   0   --   0
       RIMPL  |  A+B'  0   --   0

CC.EQ([F^?n] [CO] S^?s)
     (R)
  :select ?ops
  -> S[]:?ops |    R
     -------------------
       EQ     | NOR_F
       NEQ    | OR_F
       LT     | CO'
       LEQ    | CO'
       GT     | CO
       GEQ    | CO
       ZEROP  | NOR_F
       ONEP   | NOR_F
     let NOR_F := NOR(F^?n)
         OR_F  :=  OR(F^?n);
```

```
CC.MX(S^?s)
     (Z)
  :select ?ops
  -> for ?op in ?ops as ?i from 1
       let S.?i := S[]:?op
     let Z := OR(S.1..length(?ops))
```

A.1.9 SHFT.SYN

The file SHFT.SYN defines an n-bit shifter that can shift its inputs by m-bits to the left or right (for $m < n$). A shifter has four operations: shift left, shift right, pass inputs, pass zeros.

```
/* SHFT.SYN */

COMPONENT TYPES:

SHFT(A^?n IR^?m IL^?m S^?p)
    (Z^?n)
  :ctrl S[] (:keys (1..4))

DESIGN RULES:

SHFT(A^?n IR^?m IL^?m S^?p)
    (Z^?n)
  where ?n > ?m
  -> for ?i from 1 to ?n
       MUX(<A[?i] R.?i L.?i GND> S^?p)
          (Z[?i])
     for ?i from 1 to ?m
       let L.?i := IL[?i]
     for ?i from ?m+1 to ?n
       let L.?i := A[?i-?m]
     for ?i from 1 to ?n-?m
       let R.?i := A[?i+?m]
     for ?i from ?n-?m+1 to ?n as ?j from 1
       let R.?i := IR[?j]
```

A.1.10 MEMORY.SYN

The file MEMORY.SYN defines flip flops and n-bit registers. There are four styles of flip flops: *D*, *T*, *JK*, and *SR*, each of which can optionally have a asynchronous set and reset inputs; a D-style flip flop can also have an asynchronous load. Registers are defined in terms of D flip flops. A register can optionally have asynchronous or synchronous load, set, and reset inputs.

```
/* MEMORY.SYN */

COMPONENT TYPES:

FFLOP(I CLK [ARESET] [ASET] [ALOAD])
     (Q [N])
  :clock CLK
  :style D

FFLOP(I J CLK [ARESET] [ASET])
     (Q [N])
  :clock CLK
  :style {SR|JK}

FFLOP(I CLK [ARESET] [ASET])
     (Q [N])
  :clock CLK
  :style T

REG(I^?n CLK [ARESET] [ASET] [ALOAD])
   (Q^?n [N^?n])
  :clock CLK

REG(I^?n CLK S^?s [ARESET] [ASET] [ALOAD])
   (Q^?n [N^?n])
  :clock CLK
  :ctrl S[] (:keys ?ops)
  :operations ?ops
  where ?ops in (LOAD RESET SET)

/* serial shift register; SI = serial input from right */

REG(I^?n CLK SI S^?s [ARESET] [ASET] [ALOAD])
   (Q^?n [N^?n])
  :clock CLK
  :ctrl S[] (:keys ?ops)
  :operations ?ops
  where ?ops in (LOAD SHFT RESET SET)
      & member(SHFT, ?ops)

/* shift register m-bits left/right */

REG(I^?n CLK IR^?m IL^?m S^?s [ARESET] [ASET] [ALOAD])
   (Q^?n [N^?n])
  :clock CLK
  :ctrl S[] (:keys ?ops)
  :operations ?ops
  where ?ops in (LOAD SHL SHR RESET SET)
      & (SHL SHR) in ?ops
```

```
DESIGN RULES:

FFLOP(D CLK [ARESET] [ASET] [ALOAD])
     (Q [N])
  :clock CLK
  :style D
  -> let Q := !!(!!(D*CLK) * (ARESET+ALOAD*D+N))
        N := !!(!!(D'*CLK) * (ASET+ALOAD*D'+Q))

FFLOP(J K CLK [ARESET] [ASET] [ALOAD])
     (Q [N])
  :clock CLK
  :style JK
  -> let Q := !!(K*CLK*Q + N + ARESET)
        N := !!(J*CLK*N + Q + ASET)

FFLOP(R S CLK [ARESET] [ASET] [ALOAD])
     (Q [N])
  :clock CLK
  :style RS
  -> let Q := !!(R*CLK + N + ARESET)
        N := !!(S*CLK + Q + ASET)

FFLOP(T CLK [ARESET] [ASET])
     (Q [N])
  :clock CLK
  :style T
  -> let Q := !!(Q*T*CLK + N + ARESET)
        N := !!(N*T*CLK + Q + ASET)

REG(I^?n CLK [ARESET] [ASET] [ALOAD])
   (Q^?n N^?n)
 -> for ?i from 1 to ?n
      FFLOP(I[?i] CLK ARESET ASET ALOAD)
           (Q[?i] N[?i])
       :style D;

REG(I^?n CLK [ARESET] [ASET] [ALOAD])
   (Q^?n)
  -> for ?i from 1 to ?n
       FFLOP(I[?i] CLK ARESET ASET ALOAD)
            (Q[?i])
        :style D;
```

```
REG(I^?n CLK S [ARESET] [ASET] [ALOAD])
   (Q^?n N^?n)
  :ctrl S (:keys ?ops)
  :operations ?ops
  where ?ops = (LOAD)
  -> for ?i from 1 to ?n
       {
         FFLOP(D.?i CLK ARESET ASET ALOAD)
              (Q[?i] N[?i])
          :style D
         let D.?i := I[?i]*S + Q[?i]*S'
       }
REG(I^?n CLK S [ARESET] [ASET] [ALOAD])
   (Q^?n)
  :ctrl S (:keys ?ops)
  :operations ?ops
  where ?ops = (LOAD)
  -> for ?i from 1 to ?n
       {
         FFLOP(D.?i CLK ARESET ASET ALOAD)
              (Q[?i])
          :style D
         let D.?i := I[?i]*S + Q[?i]*S'
       }
```

A.1.11 CNTR.SYN

The file CNTR.SYN defines an n-bit up/down counter with optional asynchronous clear.

```
/* CNTR.SYN */
COMPONENT TYPES:
CNTR(I^?n CLK [CLR] LOAD UP DN)
    (O^?n)
  :style {BINARY|BCD} (default: BINARY)
DESIGN RULES:
CNTR(I^?n CLK [CLR] LOAD UP DN)
    (O^?n)
   :style BINARY
   -> let DO.1 := DN*!LOAD
          U.1  := UP*!LOAD
      for ?i from 1 to ?n
       {
         let D.?i := I[?i]*LOAD + DO.?i(+)Q.?i + U.?i(+)Q.?i
         FFLOP(D.?i CLK CLR)
              (Q.?i)
           :style D
       }
    for ?i from 2 to ?n
      let DO.?i := (!Q.?i-1)*(DO.?i-1)
          U.?i := (Q.?i-1)*(U.?i-1)
    for ?i from 1 to ?n
      let Q.?i := O[?i]
```

A.2 Interfacing to GENUS

This section lists the component types and methods that are specific to interfacing DTAS to the GENUS component library for high-level synthesis.

A.2.1 GENUS.SYN

The file GENUS.SYN defines the interface from the GENUS generic components defined in Dutt (1988) to the generic components defined by the above SYN files. Components not covered are noted with comments.

```
/* GENUS.SYN */

COMPONENT TYPES:

/************************  GENERIC COMPONENTS  ************************/

/* anything that is not a particular type of component is an entity */

ENTITY(IO^?n)
      (OO^?m)
  :usage GENUS

/* unit constant */

CONSTANT()
        (OO^?n)
  :value ?val
  :usage GENUS

/************************  ARTIFACTUAL COMPONENTS  ************************/

/* input/output interface to a VHDL netlist */

INPORT(IO^?n)
      (OO^?n)
  :usage GENUS

OUTPORT(IO^?n)
       (OO^?n)
  :usage GENUS

/* splices two or more lines of varying width into a single line */

CONCAT(IO^?n)
      (OO^?n)
  :usage GENUS

/* selects some subset of input lines specified by ?i (a list of indexes
   or a single index) to connect at output */

SELECT(IO^?n)
      (OO^?m)
  :index ?i
  :usage GENUS
  where listp(?i) & ?i in (0..?n-1)

SELECT(IO^?n)
      (OO)
  :index ?i
  :usage GENUS
  where integerp(?i) & member(?i, (0..?n-1))
```

```
/************************  COMBINATION COMPONENTS  ************************/

/* (bitwise) boolean gates are covered in GATES.SYN */

AND(IO^?m:?n)
   (OO^?n)
  :usage GENUS

OR(IO^?m:?n)
  (OO^?n)
  :usage GENUS

NAND(IO^?m:?n)
    (OO^?n)
  :usage GENUS

NOR(IO^?m:?n)
   (OO^?n)
  :usage GENUS

XOR(IO^?m:?n)
   (OO^?n)
  :usage GENUS

XNOR(IO^?m:?n)
    (OO^?n)
  :usage GENUS

NOT(IO^?n)
   (OO^?n)
  :usage GENUS

/* logic unit -- defined as FG (function generator) in ALU.SYN */

LU(IO^?n I1^?n ISEL^?s)
  (OO^?n)
  :ctrl ISEL[] (:keys ?ops)
  :operations ?ops
  :usage GENUS
  where ?ops in (ZERO ONE AND NAND OR NOR XOR XNOR LID
                 RID LNOT RNOT RINHI LINHI LIMPL RIMPL)

/* (bitwise) multiplexer are covered in MUX.SYN */

MUX(IO^?m:?n ISEL^?m)
   (OO^?n)
  :ctrl ISEL[] (:keys (0..?m-1))
  :usage GENUS
```

```
/* decoders/encoders are covered in DECODE.SYN */

DECODE(IO^?n)
      (OO^?m)
  :usage GENUS

ENCODE(IO^?n)
      (OO^?m)
  :usage GENUS

/* comparator -- defined in COMPARE.SYN */

COMPAR(IO^?n I1^?n)
      ([OEQ] [ONEQ] [OGT] [OLT] [OGEQ] [OLEQ])
  :operations ?ops
  :usage GENUS
  where ?ops in (EQ NEQ GT LT GEQ LEQ)

/* shifter -- NOT YET DEFINED */

/* barrel shifter -- NOT YET DEFINED */

/* adder/subtractor -- define as special case of ALU */

ADD(IO^?n I1^?n [ICIN])
   (OO^?n [OCOUT])
  :usage GENUS

/* arithmetic logic unit -- defined in ALU.SYN */

ALU(IO^?n I1^?n [ICIN] ISEL^?s)
   (OO^?n [OCOUT] [OREL] [OZERO] [OSIGN] [OVFLOW])
  :ctrl ISEL[] (:keys ?ops)
  :operations ?ops
  :usage GENUS
  where ?ops in (ADD SUB INC DEC
                 /* ADD1P SUB1P ADD1M SUB1M -- NOT DEFINED */
                 EQ NEQ GT LT GEQ LEQ ZEROP ONEP
                 ZERO ONE AND NAND OR NOR XOR XNOR LID
                 RID LNOT RNOT RINHI LINHI LIMPL RIMPL)

/* multipliers are covered by MULT.SYN */

MULT(IO^?n I1^?n)
    (OO^?n)
  :usage GENUS

MULT(IO^?n I1^?m)
    (OO^?n+?m)
  :usage GENUS

/* dividers -- NOT YET DEFINED */
```

```
/*********************  SEQUENTIAL COMPONENTS  ************************/

/* registers -- defined in REG.SYN */

REGI(IO^?n [ARESET] [ASET] ISEL^?s CLK)
    (OQ^?n [OQN^?n])
  :ctrl ISEL[] (:keys ?ops)
  :operations ?ops
  :usage GENUS
  where ?ops in (LOAD SHL SHR RESET SET)

REGI(IO^?n [ARESET] [ASET] CLK)
    (OQ^?n [OQN^?n])
  :usage GENUS

/* counters -- defined in CNTR.SYN */

COUNTER(IO^?n [ASET] [ARESET] CLK ISEL^?s)
       (OO^?n)
  :ctrl ISEL[] (:keys ?ops)
  :operations ?ops
  :usage GENUS
  where ?ops in (LOAD UP DOWN ZERO)

/* register file -- NOT YET DEFINED */

/* stack -- NOT YET DEFINED */

/* fifo -- NOT YET DEFINED */

/* memory -- NOT YET DEFINED */

/***************  INTERFACE AND MISCELLANEOUS COMPONENTS  *****************/

/* interface unit -- NOT YET DEFINED */

/* port -- NOT YET DEFINED */

/* bus -- NOT YET DEFINED */

/* wired-or */

WIRED_OR(IO^?m:?n)
        (OO^?n)
  :usage GENUS

/* extract -- NOT YET DEFINED */

/* clock generator */

CLOCK_GEN()
         (OO)
  :usage GENUS

/* delay -- NOT YET DEFINED */
```

```
/***************************** DESIGN RULES ******************************/

DESIGN RULES:

ENTITY(IO^?n)
      (OO^?m)
  :usage GENUS
  -> for ?i from 1 to ?n
       let IO[?i] := GND
     for ?i from 1 to ?m
       let OO[?i] := GND;

CONSTANT()
       (OO^?n)
  :value ?val
  :usage GENUS
  -> for ?i from 1 to ?n
       let OO[?i] := GND;

INPORT(IO^?n)
      (OO^?n)
  :usage GENUS
  -> for ?i from 1 to ?n
       let OO[?i] := IO[?i];

OUTPORT(IO^?n)
       (OO^?n)
  :usage GENUS
  -> for ?i from 1 to ?n
       let OO[?i] := IO[?i];

CONCAT(IO^?n)
      (OO^?n)
  :usage GENUS
  -> for ?i from 1 to ?n
       let OO[?i] := IO[?i];

SELECT(IO^?n)
      (OO^?m)
  :index ?l
  :usage GENUS
  where listp(?l) & ?l in (0..?n-1)
  -> for ?i from 1 to ?m as ?j in ?l
       let OO[?i] := IO[?j+1];

SELECT(IO^?n)
      (OO)
  :index ?i
  :usage GENUS
  where integerp(?i) & member(?i, (0..?n-1))
  -> let OO := IO[?i+1];
```

```
AND(IO^?m:?n)
   (OO^?n)
  :usage GENUS
  -> AND(IO^?m:?n)
        (OO^?n);

OR(IO^?m:?n)
  (OO^?n)
  :usage GENUS
  -> OR(IO^?m:?n)
       (OO^?n);

NAND(IO^?m:?n)
    (OO^?n)
  :usage GENUS
  -> NAND(IO^?m:?n)
         (OO^?n);

NOR(IO^?m:?n)
   (OO^?n)
  :usage GENUS
  -> NOR(IO^?m:?n)
        (OO^?n);

XOR(IO^?m:?n)
   (OO^?n)
  :usage GENUS
  -> XOR(IO^?m:?n)
        (OO^?n);

XNOR(IO^?m:?n)
    (OO^?n)
  :usage GENUS
  -> XNOR(IO^?m:?n)
         (OO^?n);

NOT(IO^?n)
   (OO^?n)
  :usage GENUS
  -> INV(IO^?n)
        (OO^?n);

LU(IO^?n I1^?n ISEL^?s)
  (OO^?n)
  :operations ?ops
  :usage GENUS
  where ?ops in (ZERO ONE AND NAND OR NOR XOR XNOR LID
                 RID LNOT RNOT RINHI LINHI LIMPL RIMPL)
  -> FG(IO^?n I1^?n ISEL^?s)
       (OO^?n)
       :operations ?ops;
```

```
MUX(IO^?m:?n ISEL^?m)
   (OO^?n)
  :usage GENUS
  -> MUX(IO^?m:?n S[] => ISEL[1..?m])
        (OO^?n)
       :ctrl S[];

DECODE(IO^?n)
      (OO^?m)
  :usage GENUS
  -> DECODE(IO^?n)
           (OO^?m)
       :style BINARY;

ENCODE(IO^?n)
      (OO^?m)
  :usage GENUS
  -> ENCODE(IO^?n)
           (OO^?m)
       :style BINARY;

COMPAR(IO^?n I1^?n)
      ([OEQ] [ONEQ] [OGT] [OLT] [OGEQ] [OLEQ])
  :operations ?ops
  :usage GENUS
  where ?ops in (EQ NEQ GT LT GEQ LEQ)
  -> let ?r := length(?ops)
     COMPARE(IO^?n I1^?n)
            (R^?r)
       :operations ?ops
     for ?i from ?r by -1 as ?op in ?ops
       if ?op = EQ
         let R[?i] := OEQ
       else
       if ?op = NEQ
         let R[?i] := ONEQ
       else
       if ?op = GT
         let R[?i] := OGT
       else
       if ?op = LT
         let R[?i] := OLT
       else
       if ?op = GEQ
         let R[?i] := OGEQ
       else
       if ?op = LEQ
         let R[?i] := OLEQ;
```

```
ADD(IO^?n I1^?n [OCIN])
   (OO^?n [OCOUT])
  :usage GENUS
  -> ADD(IO^?n I1^?n OCIN)
        (OO^?n OCOUT);

ALU(IO^?n I1^?n [OCIN] ISEL^?s)
   (OO^?n [OCOUT] [OREL])
  :operations ?ops
  :usage GENUS
  where ?ops in (ADD SUB INC DEC
                 /* ADD1P SUB1P ADD1M SUB1M -- NOT DEFINED */
                 EQ NEQ GT LT GEQ LEQ ZEROP ONEP
                 ZERO ONE AND NAND OR NOR XOR XNOR LID
                 RID LNOT RNOT RINHI LINHI LIMPL RIMPL)
  -> ALU(IO^?n I1^?n OCIN ISEL^?s)
        (OO^?n OCOUT OREL)
       :operations ?ops;

MULT(IO^?n I1^?n)
    (OO^?n)
  :usage GENUS
  -> MULT(IO^?n I1^?n)
        (OO^?n);

MULT(IO^?n I1^?m)
    (OO^?n+?m)
  :usage GENUS
  -> MULT(IO^?n I1^?n)
        (OO^?n+?m);
```

```
REGI(IO^?n [ARESET] [ASET] ISEL^?s CLK)
    (OQ^?n OQN^?n)
  :ctrl ISEL[] (:keys ?ops)
  :operations ?ops
  :usage GENUS
  -> REG(IO^?n CLK ISEL^?s ARESET ASET)
       (OQ^?n OQN^?n)
      :operations ?ops;

REGI(IO^?n [ARESET] [ASET] ISEL^?s CLK)
    (OQ^?n)
  :ctrl ISEL[] (:keys ?ops)
  :operations ?ops
  :usage GENUS
  -> REG(IO^?n CLK ISEL^?s ARESET ASET)
       (OQ^?n)
      :operations ?ops;

REGI(IO^?n [ARESET] [ASET] CLK)
    (OQ^?n [OQN^?n])
  :usage GENUS
  -> REG(IO^?n CLK ARESET ASET)
       (OQ^?n OQN^?n);

REGI(IO^?n [ARESET] [ASET] CLK)
    (OQ^?n)
  :usage GENUS
  -> REG(IO^?n CLK ARESET ASET)
       (OQ^?n);

COUNTER(IO^?n [ASET] [ARESET] CLK ISEL^?s)
      (OO^?n)
  :operations ?ops
  :usage GENUS
  where ?ops in (LOAD UP DOWN ZERO)
  -> CNTR(IO^?n <ASET ARESET> CLK ISEL^?s)
       (OO^?n)
      :operations ?ops;
```

A.3 Library-Specific SYN Files

This section lists the SYN files needed to map designs into one of the two cell libraries used in the experiments presented in Chapter 7: an MCNC benchmark library and a macrocell library from LSI Logic, Inc.

A.3.1 MCNC.SYN

The file MCNC.SYN defines the methods necessary to map designs into the MCNC standard cell benchmark library "lib2.mis2lib" (Lisanke, 1988). This library contains three different inverters, 2-, 3-, and 4-input NAND and NOR gates, 2-input XOR and XNOR gates, 16 different two-level Boolean gates.

```
/* MCNC.SYN */

DESIGN RULES:

AND(I^2)
   (O)
  -> NAND(I^2)
         (Z)
      INV(Z)
         (O);

AND(I^3)
   (O)
  -> NAND(I^3)
         (Z)
      INV(Z)
         (O);

AND(I^4)
   (O)
  -> NAND(I^4)
         (Z)
      INV(Z)
         (O);

OR(I^2)
  (O)
  -> NOR(I^2)
         (Z)
      INV(Z)
         (O);

OR(I^3)
  (O)
  -> NOR(I^3)
         (Z)
      INV(Z)
         (O);

OR(I^4)
  (O)
  -> NOR(I^4)
         (Z)
      INV(Z)
         (O);
```

```
MUX(I^2 S^1)
   (O)
  -> GATE(I[1] S.1 I[2] S.2)
         (Z)
        :name AOI22
        :source LIBRARY
      let S.1 := S[]:1
          S.2 := S[]:2
          O := !Z;
```

A.3.2 LSI.SYN

The file LSI.SYN defines the component types and methods necessary to map designs into a subset of LSI Logic, Inc.'s macrocell library consisting of Boolean cells and complex functional cells, including two 1-bit adder cells (FA1 and FA1A), a 4-bit adder (FA4) and two related carry look-ahead generators (CLA1 and CLA2), and 2-to-1, 4-to-2, and 8-to-4 multiplexers. This library is drawn from the cells catalogued in a databook of 2-micron cells (LSI, 1987).

```
/* LSI.SYN */

COMPONENT TYPES:

ADD(X^?n Y^?n CI)
   (S^?n CO)
  :data X[] Y[] S[]
  :carry CI CO
  :style LSI

ADD(A^4 B^4 C0 G0 G1)
   (S^4 C4 P3)
  :data A[] B[] S[]
  :carry C0 C4
  :cgenr G0 G1
  :cprop P3

CLA(A^4 B^4 C0)
   (G0 G1 C4)
  :data A[] B[]
  :carry C0 C4
  :cgenr G0 G1

DESIGN RULES:

ADD(X^?n Y^?n CI)
   (S^?n CO)
  where ?n > 1
  ->
     ADD(X^?n Y^?n CI)
        (S^?n CO)
       :style LSI;

ADD(X^?n Y^?n)
   (S^?n CO)
  where ?n > 1
  ->
     ADD(X^?n Y^?n GND)
        (S^?n CO)
       :style LSI;

ADD(X^?n Y^?n)
   (S^?n)
  where ?n > 1
  ->
     ADD(X^?n Y^?n GND)
        (S^?n GND)
       :style LSI;
```

```
ADD(X^3 Y^3 CI)
   (S^3 CO)
  :style LSI
  ->
     ADD(<X[1..3] GND> <Y[1..3] GND> CI G0 G1)
        (<S[1..3] GND> CO GND)
       :source LIBRARY
     let G0 := X[1]*Y[1]
         G1 := X[2]*Y[2];

ADD(X^4 Y^4 CI)
   (S^4 CO)
  :style LSI
  ->
     ADD(X[1..4] Y[1..4] CI G0 G1)
        (S[1..4] C4 P3)
       :source LIBRARY
     let G0 := X[1]*Y[1]
         G1 := X[2]*Y[2]
         CO := ((C4*P3)'*(X[4]*Y[4])')';

ADD(X^?n Y^?n CI)
   (S^?n [CO])
  :style LSI
  where ?n > 4
  ->
     ADD(X[1..4] Y[1..4] CI G0.1 G1.1)
        (S[1..4] GND GND)
       :source LIBRARY
     CLA(X[1..4] Y[1..4] CI)
        (G0.1 G1.1 C4.1)
       :source LIBRARY
       :name CLA1
     let ?s := ceiling(?n/4) - 1
     for ?i from 2 to ?s as ?j from 5 by 4
       {
         ADD(X[?j..?j+3] Y[?j..?j+3] C4.?i-1 G0.?i G1.?i)
            (S[?j..?j+3] GND GND)
           :source LIBRARY
         CLA(X[?j..?j+3] Y[?j..?j+3] C4.?i-1)
            (G0.?i G1.?i C4.?i)
           :source LIBRARY
           :name CLA2
       }
     let ?r := 4*?s + 1
     ADD(X[?r..?n] Y[?r..?n] C4.?s)
        (S[?r..?n] CO);
```

```
MUX(I^2:3 S^?s)
   (O^3)
  :ctrl S[] (:keys (1 2))
  -> MUX(I[1..2:1..2] SEL)
        (O[1..2])
       :source LIBRARY
     MUX(I[1..2:3] SEL)
        (O[3])
       :source LIBRARY
     let SEL := S[]:2;

MUX(I^2:?n S^?s)
   (O^?n)
  :ctrl S[] (:keys (1 2))
  where ?n > 4
  -> let ?r := mod(?n, 4)
     for ?i from 1 to ?n - ?r by 4
       MUX(I[1..2:?i..?i+3] SEL)
          (O[?i..?i+3])
         :source LIBRARY
     let SEL := S[]:2
     if ?r > 0
       MUX(I[1..2:?n-?r+1..?n] S^?s)
          (O[?n-?r+1..?n]);

AU(A^?n B^?n [CI.1] S^?s)
  (F^?n [CO])
  :operations ?ops
  where ?n > 2 & setequalp(?ops, (ADD SUB))
  -> let ?r := mod(?n,2)
     CC.CI(CI.1 S^?s)
          (CI.2)
       :select ?ops
     for ?i from 1 to ?n-?r by 2
       AU(A[?i..?i+1] B[?i..?i+1] C.?i SUB)
         (F[?i..?i+1] C.?i+2)
        :operations (ADD SUB)
        :source LIBRARY
     if ?r = 1
       {
         ADD(A[?n] B C.?n)
            (F[?n] C.?n+1)
         let B := SUB(+)B[?n]
       }
     let SUB := S[]:SUB
         C.1 := CI.2
         C.?n+1 := CO;
```

Appendix B
The LIB Files

In the experiments described in Chapter 7, DTAS mapped designs into three layout cell libraries:

1. The MCNC standard cell benchmark library "lib2.mis2lib" (Lisanke, 1988). This library contains three different inverters, 2-, 3-, and 4-input NAND and NOR gates, 2-input XOR and XNOR gates, 16 different two-level Boolean gates.
2. A subset of LSI Logic, Inc.'s macrocell library consisting of Boolean cells only. This library is drawn from the cells catalogued in a databook of 2-micron cells (LSI, 1987).
3. A subset of LSI Logic, Inc.'s macrocell library consisting of Boolean cells and complex functional cells, including two 1-bit adder cells (FA1 and FA1A), a 4-bit adder (FA4) and two related carry look-ahead generators (CLA1 and CLA2), and 2-to-1, 4-to-2, and 8-to-4 multiplexers. This library is also drawn from the cells catalogued in a databook of 2-micron cells (LSI, 1987).

This appendix contains a complete listing of the DTAS LIB files that define these libraries. The first two libraries are described in MISII format (Lisanke, 1988); the third is described in DTAS LIB file syntax.

B.1 The MCNC Library

The file lib2.mis2lib is defined as shown below.

```
GATE inv1x   928.00 O = ! a;
        PIN a INV 0.0514 999.0 0.4200 4.7100 0.4200 3.6000
GATE inv2x   928.00 O = ! a;
        PIN a INV 0.1009 999.0 0.3000 1.9800 0.2900 1.8200
GATE inv4x  1392.00 O = ! a;
        PIN a INV 0.1897 999.0 0.2300 1.0800 0.2700 0.8500
GATE xor  2320.00 O = ((!a * b) + (a * !b));
        PIN a UNKNOWN 0.1442 999.0 1.7700 5.2300 0.9600 4.6400
        PIN b UNKNOWN 0.1381 999.0 1.9400 4.6500 1.1400 5.2200
GATE xnor  2320.00 O = ((!a * !b) + (a * b));
        PIN a UNKNOWN 0.1502 999.0 1.1100 4.8600 1.0700 3.3900
        PIN b UNKNOWN 0.1352 999.0 1.5500 4.8700 1.0700 3.3900
GATE nand2  1392.00 O = !(a * b);
        PIN a INV 0.0777 999.0 0.6400 4.0900 0.4000 2.5700
        PIN b INV 0.0716 999.0 0.4600 4.1000 0.3700 2.5700
GATE nand3  1856.00 O = !(a * b * c);
        PIN a INV 0.1000 999.0 0.8900 3.6000 0.5100 2.4900
        PIN b INV 0.0828 999.0 0.7100 4.1100 0.4200 2.5000
        PIN c INV 0.0777 999.0 0.5600 4.3900 0.3500 2.4900
GATE nand4  2320.00 O = !(a * b * c * d);
        PIN a INV 0.1030 999.0 1.2700 3.6200 0.6700 2.3900
        PIN b INV 0.0980 999.0 1.0900 3.6100 0.6100 2.3900
        PIN c INV 0.0980 999.0 0.8200 3.6200 0.5500 2.4000
        PIN d INV 0.1050 999.0 0.5800 3.6200 0.3800 2.3900
GATE nor2  1392.00 O = !(a + b);
        PIN a INV 0.0736 999.0 0.3300 3.6400 0.4500 3.6400
        PIN b INV 0.0968 999.0 0.5000 3.6400 0.7000 3.6600
GATE nor3  1856.00 O = !(a + b + c);
        PIN a INV 0.0856 999.0 0.8400 5.0400 1.3000 3.4500
        PIN b INV 0.0806 999.0 0.7800 5.0300 1.1400 3.4300
        PIN c INV 0.0826 999.0 0.5200 5.0300 0.8400 3.4400
GATE nor4  2320.00 O = !(a + b + c + d);
        PIN a INV 0.0887 999.0 0.4100 5.9100 1.1600 3.2000
        PIN b INV 0.0867 999.0 0.8500 5.9100 1.5300 3.1800
        PIN c INV 0.0867 999.0 1.1100 5.9200 1.7500 3.1900
        PIN d INV 0.0887 999.0 1.2700 5.9100 1.9400 3.2000
GATE aoi21  1856.00 O = !((a1 * a2) + b);
        PIN a1 INV 0.1029 999.0 0.7500 3.5200 0.6700 2.5300
        PIN a2 INV 0.0908 999.0 0.6700 3.6400 0.6200 2.5200
        PIN b INV 0.1110 999.0 0.5800 3.6400 0.2100 1.2800
```

```
GATE aoi31  2320.00 O =  !((a1 * a2 * a3) + b);
        PIN a1 INV 0.1009 999.0 0.9100 4.0400 0.8100 2.8600
        PIN a2 INV 0.1049 999.0 1.0500 3.9300 0.8700 2.8700
        PIN a3 INV 0.1059 999.0 1.1500 3.9400 0.9400 2.8600
        PIN b INV 0.0979 999.0 0.8900 4.0600 0.2500 1.2800
GATE aoi22  2320.00 O =  !((a1 * a2) + (b1 * b2));
        PIN a1 INV 0.1019 999.0 0.9200 3.4600 0.9400 2.7900
        PIN a2 INV 0.0908 999.0 0.8400 3.6400 0.8500 2.7900
        PIN b1 INV 0.0958 999.0 0.6100 3.6400 0.4900 2.9300
        PIN b2 INV 0.0988 999.0 0.7000 3.6400 0.5400 2.9300
GATE aoi32  2784.00 O =  !((a1 * a2 * a3) + (b1 * b2));
        PIN a1 INV 0.1029 999.0 1.0600 3.8100 0.9600 2.9100
        PIN a2 INV 0.1009 999.0 1.2000 3.8100 1.0300 2.9000
        PIN a3 INV 0.1060 999.0 1.2900 3.6900 1.0600 2.9100
        PIN b1 INV 0.0979 999.0 0.9100 3.8100 0.4300 2.1200
        PIN b2 INV 0.1049 999.0 0.7800 3.5900 0.4300 2.1200
GATE aoi33  3248.00 O =  !((a1 * a2 * a3) + (b1 * b2 * b3));
        PIN a1 INV 0.1029 999.0 1.3300 3.9100 1.3000 2.9100
        PIN a2 INV 0.1029 999.0 1.4600 3.8400 1.4100 2.9100
        PIN a3 INV 0.1120 999.0 1.4700 3.6500 1.4100 2.9100
        PIN b1 INV 0.1029 999.0 1.1100 3.5900 0.7600 2.9000
        PIN b2 INV 0.0949 999.0 1.0400 3.9100 0.6800 2.9100
        PIN b3 INV 0.1039 999.0 0.8400 3.5800 0.6400 2.9000
GATE aoi211  2320.00 O = !((a1 * a2) + b + c);
        PIN a1 INV 0.1039 999.0 1.1200 4.8100 1.0300 2.3800
        PIN a2 INV 0.1090 999.0 1.2900 4.8100 1.0300 2.3800
        PIN b INV 0.1080 999.0 1.0400 4.8300 0.5200 1.4000
        PIN c INV 0.1008 999.0 0.6800 4.8300 0.5100 1.7900
GATE aoi221  2784.00 O = !((a1 * a2) + (b1 * b2) + c);
        PIN a1 INV 0.1089 999.0 1.4800 4.4300 1.3300 2.7800
        PIN a2 INV 0.0948 999.0 1.4200 4.5600 1.4000 2.7500
        PIN b1 INV 0.1029 999.0 0.7600 4.4700 0.7900 2.8900
        PIN b2 INV 0.1049 999.0 0.7300 4.5800 0.7800 2.9100
        PIN c INV 0.1110 999.0 1.3900 4.5600 0.7000 1.5100
GATE aoi222  3712.00 O = !((a1 * a2) + (b1 * b2) + (c1 * c2) );
        PIN a1 INV 0.1019 999.0 1.7700 4.5800 1.5600 2.9500
        PIN a2 INV 0.0958 999.0 1.7300 4.6900 1.6000 2.9300
        PIN b1 INV 0.1039 999.0 1.3400 4.6800 1.2100 2.9200
        PIN b2 INV 0.1039 999.0 1.5000 4.6900 1.2200 2.9200
        PIN c1 INV 0.0958 999.0 0.9200 4.6700 0.8100 2.9200
        PIN c2 INV 0.1039 999.0 0.7700 4.4700 0.7600 2.9200
GATE oai21  1856.00 O = !( (a1 + a2) * b);
        PIN a1 INV 0.1019 999.0 0.6900 3.9400 0.5300 2.4700
        PIN a2 INV 0.0979 999.0 0.8700 3.9300 0.6300 2.4700
        PIN b INV 0.0998 999.0 0.3700 2.0500 0.5700 2.5100
```

```
GATE oai31  2320.00 O = !( (a1 + a2 + a3) * b);
        PIN a1 INV 0.1089 999.0 1.2700 4.7100 1.0300 2.4300
        PIN a2 INV 0.1049 999.0 1.1100 4.7100 1.0400 2.5700
        PIN a3 INV 0.1090 999.0 0.8500 4.7100 0.6900 2.3800
        PIN b INV 0.1059 999.0 0.3800 1.8600 0.8100 2.7300
GATE oai22  2320.00 O = !( (a1 + a2) * (b1 + b2));
        PIN a1 INV 0.1009 999.0 1.1000 4.0600 0.9000 2.5000
        PIN a2 INV 0.1029 999.0 0.9900 4.0600 0.6800 2.3600
        PIN b1 INV 0.0958 999.0 0.6900 3.6600 0.7400 2.5300
        PIN b2 INV 0.1039 999.0 0.6100 3.6600 0.5600 2.0600
GATE oai32  2784.00 O = !( (a1 + a2 + a3) * (b1 + b2));
        PIN a1 INV 0.1130 999.0 1.3900 4.4600 1.0400 2.4600
        PIN a2 INV 0.1069 999.0 1.2500 4.4600 1.0900 2.6300
        PIN a3 INV 0.1140 999.0 0.9900 4.4600 0.7400 2.4200
        PIN b1 INV 0.1059 999.0 0.5800 3.2000 0.7900 2.7100
        PIN b2 INV 0.1130 999.0 0.6800 3.2100 0.8300 2.3400
GATE oai33  3248.00 O = !( (a1 + a2 + a3) * (b1 + b2 + b3));
        PIN a1 INV 0.1170 999.0 1.5800 4.3000 1.4800 2.4700
        PIN a2 INV 0.1089 999.0 1.5000 4.3100 1.4200 2.6300
        PIN a3 INV 0.1079 999.0 1.2400 4.3100 1.1700 2.6500
        PIN b1 INV 0.1170 999.0 0.8000 4.3000 0.8200 2.2700
        PIN b2 INV 0.1089 999.0 0.0000 4.3000 1.1700 2.6400
        PIN b3 INV 0.1109 999.0 1.1300 4.3100 1.3500 2.6500
GATE oai211  2320.00 O = !( (a1 + a2) * b * c);
        PIN a1 INV 0.1070 999.0 1.1200 4.1700 0.5900 2.3100
        PIN a2 INV 0.1131 999.0 1.3000 4.1600 0.7900 2.3600
        PIN b INV 0.1050 999.0 0.5100 2.1300 0.6900 2.4000
        PIN c INV 0.1050 999.0 0.5000 2.4600 0.5200 2.4100
GATE oai221  2784.00 O = !( (a1 + a2) * (b1 + b2) * c);
        PIN a1 INV 0.1039 999.0 1.5800 4.1700 1.1100 2.4700
        PIN a2 INV 0.1050 999.0 1.4800 4.1700 0.8600 2.3600
        PIN b1 INV 0.1080 999.0 0.9400 4.0300 0.8100 2.5000
        PIN b2 INV 0.1060 999.0 0.7600 4.0300 0.6400 2.5000
        PIN c INV 0.1019 999.0 0.7800 2.2800 0.9000 2.5400
GATE oai222  3248.00 O = !( (a1 + a2) * (b1 + b2) * (c1 + c2));
        PIN a1 INV 0.1161 999.0 1.7700 3.7500 1.2100 2.4700
        PIN a2 INV 0.1110 999.0 1.6200 3.7500 1.1300 2.4800
        PIN b1 INV 0.1009 999.0 1.1700 3.5800 1.0700 2.4800
        PIN b2 INV 0.1191 999.0 1.3500 3.5800 1.1000 2.3500
        PIN c1 INV 0.1060 999.0 0.9900 3.5900 0.9300 2.4900
        PIN c2 INV 0.1140 999.0 0.8200 3.5800 0.7900 2.4800
GATE zero       0       O=CONST0;
GATE one        0       O=CONST1;
```

B.2 The LSI Logic Library (Subset I)

The file lsi.mis2lib is defined as shown below.

```
GATE AN2        2       Z=A*B;
        PIN * NONINV  1 999 0.48 0.1443 0.77 0.0523

GATE AN3        2       Z=A*B*C;
        PIN * NONINV  1 999 0.69 0.1458 0.85 0.0589

GATE AN4        3       Z=A*B*C*D;
        PIN * NONINV  1 999 0.97 0.1523 0.95 0.0589

GATE AO1        2       Z=!((A*B)+C+D);
        PIN * INV     1 999 1.11 0.3864 0.27 0.0824

GATE AO2        2       Z=!((A*B)+(C*D));
        PIN * INV     1 999 0.82 0.2612 0.47 0.0824

GATE AO3        2       Z=!((A+B)*C*D);
        PIN * INV     1 999 0.52 0.2612 0.39 0.1136

GATE AO4        2       Z=!((A+B)*(C+D));
        PIN * INV     1 999 0.92 0.2612 0.37 0.0824

GATE AO5        3       Z=!((A*B)+(A*C)+(B*C));
        PIN A INV     2 999 1.12 0.2612 0.45 0.0788
        PIN B INV     2 999 1.12 0.2612 0.45 0.0788
        PIN C INV     1 999 1.12 0.2612 0.45 0.0788

GATE AO6        2       Z=!((A*B)+C);
        PIN * INV     1 999 0.82 0.2612 0.27 0.0824

GATE AO7        2       Z=!((A+B)*C);
        PIN * INV     1 999 0.52 0.2612 0.37 0.0824

GATE EN         3       Z=A*B+!A*!B;
        PIN A UNKNOWN 1 999 0.79 0.1458 1.06 0.0653
        PIN B UNKNOWN 2 999 0.79 0.1458 1.06 0.0653

GATE EN         3       Z=!(!A*B+A*!B);
        PIN A UNKNOWN 1 999 0.79 0.1458 1.06 0.0653
        PIN B UNKNOWN 2 999 0.79 0.1458 1.06 0.0653

GATE EN3        7       Z=A*B*!C+A*!B*C+!A*B*C+!A*!B*!C;
        PIN A UNKNOWN 1 999 1.68 0.1517 1.85 0.0790
        PIN B UNKNOWN 3 999 1.68 0.1517 1.85 0.0790
        PIN C UNKNOWN 2 999 1.68 0.1517 1.85 0.0790

GATE EN3        7       Z=!(A*B*C+A*!B*!C+!A*B*!C+!A*!B*C);
        PIN A UNKNOWN 1 999 1.68 0.1517 1.85 0.0790
        PIN B UNKNOWN 3 999 1.68 0.1517 1.85 0.0790
        PIN C UNKNOWN 2 999 1.68 0.1517 1.85 0.0790
```

```
GATE EO          3       Z=!A*B+A*!B;
        PIN A UNKNOWN 1 999 0.79 0.1458 1.06 0.0653
        PIN B UNKNOWN 2 999 0.79 0.1458 1.06 0.0653

GATE EO          3       Z=!(A*B+!A*!B);
        PIN A UNKNOWN 1 999 0.79 0.1458 1.06 0.0653
        PIN B UNKNOWN 2 999 0.79 0.1458 1.06 0.0653

GATE EO1         3       Z=!(A*B+!(C+D));
        PIN A INV     1 999 0.82 0.2612 0.97 0.0839
        PIN B INV     1 999 0.82 0.2612 0.97 0.0839
        PIN C NONINV  1 999 0.82 0.2612 0.97 0.0839
        PIN D NONINV  1 999 0.82 0.2612 0.97 0.0839

GATE EO3         7       Z=A*B*C+A*!B*!C+!A*B*!C+!A*!B*C;
        PIN A UNKNOWN 1 999 1.68 0.1517 1.85 0.0790
        PIN B UNKNOWN 3 999 1.68 0.1517 1.85 0.0790
        PIN C UNKNOWN 2 999 1.68 0.1517 1.85 0.0790

GATE EO3         7       Z=!(A*B*!C+A*!B*C+!A*B*C+!A*!B*!C);
        PIN A UNKNOWN 1 999 1.68 0.1517 1.85 0.0790
        PIN B UNKNOWN 3 999 1.68 0.1517 1.85 0.0790
        PIN C UNKNOWN 2 999 1.68 0.1517 1.85 0.0790

GATE EON1        3       Z=!((A+B)*!(C*D));
        PIN A INV     1 999 0.82 0.2612 0.87 0.0839
        PIN B INV     1 999 0.82 0.2612 0.87 0.0839
        PIN C NONINV  1 999 0.82 0.2612 0.87 0.0839
        PIN D NONINV  1 999 0.82 0.2612 0.87 0.0839

GATE IV          1       Z=!A;
        PIN * INV     1 999 0.38 0.1443 0.15 0.0589

GATE IVA         1       Z=!A;
        PIN * INV   1.5 999 0.24 0.0718 0.25 0.0589

GATE ND2         1       Z=!(A*B);
        PIN * INV     1 999 0.50 0.1377 0.13 0.0854

GATE ND3         2       Z=!(A*B*C);
        PIN * INV     1 999 0.65 0.1411 0.37 0.1146

GATE ND4         2       Z=!(A*B*C*D);
        PIN * INV     1 999 0.65 0.1411 0.45 0.1411

GATE ND5         4       Z=!(A*B*C*D*E);
        PIN * INV     1 999 1.08 0.1443 1.15 0.0589

GATE ND6         5       Z=!(A*B*C*D*E*F);
        PIN * INV     1 999 0.98 0.1443 1.15 0.0589

GATE ND8         6       Z=!(A*B*C*D*E*F*G*H);
        PIN * INV     1 999 1.08 0.1443 1.45 0.0589

GATE NR2         1       Z=!(A+B);
        PIN * INV     1 999 0.55 0.2589 0.25 0.0589

GATE NR3         2       Z=!(A+B+C);
        PIN * INV     1 999 0.81 0.3864 0.25 0.0589
```

```
GATE NR4        2       Z=!(A+B+C+D);
       PIN * INV     1 999 1.07 0.5146 0.25 0.0589

GATE NR5        4       Z=!(A+B+C+D+E);
       PIN * INV     1 999 1.49 0.1458 0.87 0.0523

GATE NR6        5       Z=!(A+B+C+D+E+F);
       PIN * INV     1 999 1.59 0.1458 0.87 0.0523

GATE NR8        6       Z=!(A+B+C+D+E+F+G+H);
       PIN * INV     1 999 1.89 0.1458 0.87 0.0523

GATE OR2        1       Z=A+B;
       PIN * NONINV  1 999 0.38 0.1443 0.85 0.0589

GATE OR3        2       Z=A+B+C;
       PIN * NONINV  1 999 0.48 0.1443 1.24 0.0718

GATE OR4        2       Z=A+B+C+D;
       PIN * NONINV  1 999 0.38 0.1443 1.35 0.0788
```

B.3 The LSI Logic Library (Subset II)

The second subset of the LSI Logic macrocell library contains the Boolean cells listed above plus the functional cells listed below.

```
LIBRARY CELLS:  *** LSI Logic / CMOS Macrocell Catalogue / 1987 ***
  :DELAY nanoseconds
  :AREA  equivalent NAND gates

ADD(A B CI)
   (S CO)
  :name  FA1
  :area  10
  :load  (3 4 4)
  :delay     |          S                    CO
         ----+-------------------------------------
         A   | 1.88+0.14:1.94+0.07 0.79+0.15:1.34+0.07
         B   | 1.88+0.14:1.94+0.07 0.79+0.15:1.34+0.07
         CI  | 0.78+0.14:0.64+0.07 0.79+0.15:1.34+0.07
  -> let S   := A(+)B(+)CI
         CO  := A*B+A*CI+B*CI

ADD(A B CI)
   (S CO)
  :name  FA1A
  :area  8
  :load  (1 2 2)
  :delay     |          S                    CO
         ----+-------------------------------------
         A   | 1.99+0.15:1.86+0.07 1.89+0.15:2.14+0.07
         B   | 1.99+0.15:1.86+0.07 1.89+0.15:2.14+0.07
         CI  | 0.89+0.15:0.76+0.07 0.79+0.15:1.04+0.07
  -> let S   := A(+)B(+)CI
         CO  := A*B+A*CI+B*CI

ADD(A B)
   (S CO)
  :name  HA1
  :area  5
  :load  (2 3)
  :delay     |          S                    CO
         ----+-------------------------------------
         A   | 1.08+0.14:0.94+0.07 0.48+0.14:0.77+0.05
         B   | 1.08+0.14:0.94+0.07 0.48+0.14:0.77+0.05
  -> let S   := A(+)B
         CO  := A*B
```

```
ADD(<A0 A1> <B0 B1> C0)
   (<S0 S1> C2)
  :name  FA2
  :area  20
  :load  (3 4 3 4 4)
  :delay    |         S0                    S1                    C2
         ---+-------------------------------------------------------------
         A0 | 1.88+0.14:1.94+0.07 2.20+0.14:2.06+0.07 2.21+0.15:2.76+0.07
         B0 | 1.88+0.14:1.94+0.07 2.20+0.14:2.06+0.07 2.21+0.15:2.76+0.07
         A1 |         --          1.68+0.14:1.94+0.07 0.79+0.15:1.34+0.07
         B1 |         --          1.68+0.14:1.94+0.07 0.79+0.15:1.34+0.07
         C0 | 0.78+0.14:0.64+0.07 2.20+0.14:2.06+0.07 2.21+0.15:2.76+0.07
  -> let S0 := A0(+)B0(+)C0
         C1 := A0*B0+A0*C0+B0*C0
         S1 := A1(+)B1(+)C1
         C2 := A1*B1+A1*C1+B1*C1

CLA(<A0 A1 A2 A3> <B0 B1 B2 B3> C0)
   (G0 G1 C4)
  :name  CLA1
  :area  24
  :load  (2 2 3 4 2 2 3 4 1)
  :delay    |         G0                    G1                    C4
         ---+-------------------------------------------------------------
         A0 | 0.59+0.07:1.10+0.06         --          1.97+0.14:1.89+0.09
         A1 |         --          0.59+0.07:1.10+0.06 1.97+0.14:1.74+0.09
         A2 |         --                  --          2.12+0.14:2.04+0.09
         A3 |         --                  --          2.12+0.14:2.04+0.09
         B0 | 0.59+0.07:1.10+0.06         --          1.97+0.14:1.89+0.09
         B1 |         --          0.59+0.07:1.10+0.06 1.97+0.14:1.74+0.09
         B2 |         --                  --          2.12+0.14:2.04+0.09
         B3 |         --                  --          2.12+0.14:2.04+0.09
         C0 |         --                  --          1.97+0.14:1.89+0.09

CLA(<A0 A1 A2 A3> <B0 B1 B2 B3> C0)
   (G0 G1 C4)
  :name  CLA2
  :area  21
  :load  (2 3 2 3 2 3 2 3 1)
  :DELAY    |         G0                    G1                    C4
         ---+-------------------------------------------------------------
         A0 | 0.59+0.07:1.10+0.06         --          2.26+0.14:2.00+0.09
         A1 |         --          0.59+0.07:1.10+0.06 2.22+0.14:2.19+0.09
         A2 |         --                  --          2.63+0.14:2.64+0.09
         A3 |         --                  --          2.63+0.14:2.64+0.09
         B0 | 0.59+0.07:1.10+0.06         --          2.26+0.14:2.00+0.09
         B1 |         --          0.59+0.07:1.10+0.06 1.84+0.14:2.19+0.09
         B2 |         --                  --          2.63+0.14:2.64+0.09
         B3 |         --                  --          2.63+0.14:2.64+0.09
         C0 |         --                  --          0.72+0.14:0.77+0.09
```

```
ADD(<A0 A1 A2 A3> <B0 B1 B2 B3> C0 G0 G1)
   (<S0 S1 S2 S3> C4 P3)
  :name  FA4
  :area  50
  :delay    |          S0                    S1
        ----------------------------------------------
        A0  | 2.23+0.15:2.50+0.07 2.88+0.15:3.25+0.07
        A1  |          --         2.38+0.15:2.75+0.07
        A2  |          --                 --
        A3  |          --                 --
        B0  | 2.23+0.15:2.50+0.07 2.88+0.15:3.25+0.07
        B1  |          --         2.38+0.15:2.75+0.07
        B2  |          --                 --
        B3  |          --                 --
        G0  |          --         1.57+0.15:1.84+0.07
        G1  |          --                 --
        C0  | 0.79+0.15:1.06+0.07 1.57+0.15:1.84+0.07
            |
            |          S2                    S3
        ----------------------------------------------
        A0  | 3.25+0.15:3.52+0.07 3.83+0.15:4.10+0.07
        A1  | 3.40+0.15:3.67+0.07 4.26+0.15:4.53+0.07
        A2  | 2.23+0.15:2.50+0.07 4.11+0.15:4.38+0.07
        A3  |          --         1.99+0.15:2.16+0.07
        B0  | 3.25+0.15:3.52+0.07 3.83+0.15:4.10+0.07
        B1  | 3.40+0.15:3.67+0.07 4.26+0.15:4.53+0.07
        B2  | 2.23+0.15:2.50+0.07 4.11+0.15:4.38+0.07
        B3  |          --         1.99+0.15:2.16+0.07
        G0  | 1.78+0.15:2.05+0.07 2.67+0.15:2.84+0.07
        G1  | 1.78+0.15:2.05+0.07 1.83+0.15:2.10+0.07
        C0  | 1.81+0.15:2.08+0.07 1.65+0.15:1.92+0.07
            |
            |          P3                    C4
        ----------------------------------------------
        A0  |          --         3.04+0.14:2.98+0.09
        A1  |          --         3.47+0.14:3.19+0.09
        A2  |          --         3.32+0.14:3.12+0.09
        A3  | 1.09+0.15:1.20+0.07         --
        B0  |          --         3.04+0.14:2.98+0.09
        B1  |          --         3.47+0.14:3.19+0.09
        B2  |          --         3.32+0.14:3.12+0.09
        B3  | 1.09+0.15:1.20+0.07         --
        G0  |          --         1.98+0.14:1.91+0.09
        G1  |          --         1.14+0.14:1.00+0.09
        C0  |          --         0.86+0.14:0.86+0.09
```

```
AU(<A0 A1> <B0 B1> C0 SUB)
  (<S0 S1> C2)
  :ctrl SUB (:keys (ADD SUB))
  :operations (ADD SUB)
  :name  FAS2
  :area  26
  :delay     |          S0                    S1                    C2
         ----+-----------------------------------------------------------
         A0  | 1.88+0.14:1.94+0.07 2.20+0.14:2.06+0.07 2.01+0.15:2.26+0.07
         B0  | 1.93+0.14:1.99+0.07 2.25+0.14:2.11+0.07 2.26+0.15:2.81+0.07
         A1  |          --          1.88+0.14:1.94+0.07 0.79+0.15:1.14+0.07
         B1  |          --          1.93+0.14:1.99+0.07 0.94+0.15:1.29+0.07
         C0  | 0.78+0.14:0.64+0.07 2.20+0.14:2.06+0.07 2.21+0.15:2.76+0.07
         SUB | 1.93+0.14:1.99+0.07 2.25+0.14:2.11+0.07 2.26+0.15:2.81+0.07

MUX(<D0 D1> A)
   (Z)
  :name  MUX21H
  :area  4
  :delay    |  Z
         ---+----
         D0 | 1.2
         D1 | 1.2
         A  | 0.9
  -> A | Z
     --+---
     0 | D0
     1 | D1

MUX(<D00 D01 D10 D11> A)
   (<Z0 Z1>)
  :name  MUX22H
  :area  7
  :delay     |  Z0  Z1
         ----+--------
         D00 | 1.2 1.2
         D01 | 1.2 1.2
         D10 | 1.2 1.2
         D11 | 1.2 1.2
         A   | 0.9 0.9
  -> A |  Z0  Z1
     --+--------
     0 | D00 D01
     1 | D10 D11
```

```
MUX(<D00 D01 D02 D03 D10 D11 D12 D13> A)
   (<Z0 Z1 Z2 Z3>)
  :name  MUX24H
  :area  13
  :delay      |  Z0  Z1  Z2  Z3
              ----+----------------
              D00 | 1.2 1.2 1.2 1.2
              D01 | 1.2 1.2 1.2 1.2
              D02 | 1.2 1.2 1.2 1.2
              D03 | 1.2 1.2 1.2 1.2
              D10 | 1.2 1.2 1.2 1.2
              D11 | 1.2 1.2 1.2 1.2
              D12 | 1.2 1.2 1.2 1.2
              D13 | 1.2 1.2 1.2 1.2
              A   | 0.9 0.9 0.9 0.9
  -> A |  Z0  Z1  Z2  Z3
     --+----------------
     0 | D00 D01 D02 D03
     1 | D10 D11 D12 D13
```

Appendix C
The DAT Files

Input specifications for the experiments presented in Chapter 7 were drawn from a collection of GENUS data files. Each file contains several specifications of the GENUS component designated by the file's name; e.g., the file `mult.nl` contains specifications of a GENUS multiplier. Differences between specifications in a file is the width of their data inputs, typically varying from 4 to 64 bits. Most GENUS components have additional parameters that can effect performance; variations of these parameters are specified in separate files. Components are specified using structural VHDL. The GENUS data files are organized as outlined below.

- Boolean gates: These gates have a 1-bit output and an n-bit input, such that $n \in \{2, 4, 8, 12, 16, 32, 48, 64\}$. The files in this set include

  `and.nl or.nl nand.nl nor.nl xor.nl xnor.nl not.nl`

- Bitwise Boolean gates: A bitwise gate has an n-bit output and m n-bit inputs, such that $n \in \{2, 4, 8, 12, 16, 32, 48, 64\}$. In the first row of files, $m = 2$; in the second row, $m = 4$; and in the third row, $m = 8$. The files in this set include

  `and2.nl or2.nl nand2.nl nor2.nl xor2.nl xnor2.nl not2.nl`
  `and4.nl or4.nl nand4.nl nor4.nl xor4.nl xnor4.nl not4.nl`
  `and8.nl or8.nl nand8.nl nor8.nl xor8.nl xnor8.nl not8.nl`

- Decoder/encoder (binary): Decoders have an n-bit input and a 2^n-bit output, such that $n \in$ '2, 3, 4, 5, 6. Encoders have an n-bit output and 2^n-bit input. The files in this set include

  `decode.nl encode.nl`

- Multiplexers: Another bitwise component. Multiplexers have an n-bit output and m n-bit inputs, such that $n \in \{1, 2, 4, 8, 16, 32, 48, 64\}$. In file, `muxi.nl`, $m = i$. The files in this set include

`mux2.nl mux4.nl mux8.nl`

- Logic units: A logic unit can compute as many as 16 Boolean operations on two n-bit inputs, such that $n \in \{4, 8, 16, 32, 48, 64\}$. In file `lu.`$i$`.nl`, i operations are specified. The files in this set include

  `lu.2.nl lu.4.nl lu.8.nl lu.12.nl lu.16.nl`

- Adders (without carries) and multipliers: $n \in \{4, 8, 16, 32, 48, 64\}$. The files in this set include

  `add.nl mult.nl`

- Comparators: Comparators can compute as many as six relations on two n-bit inputs, such that $n \in \{4, 8, 16, 32, 48, 64\}$. In file `compar.`$i$`.nl`, i operations are specified. The files in this set include

  `compar.1.nl compar.2.nl compar.4.nl compar.6.nl`

- Arithmetic logic units (ALUs): ALUs can compute as many as 4 arithmetic operations, 8 comparison operations, and 16 logic operations on two n-bit inputs, such that $n \in \{4, 8, 16, 32, 48, 64\}$. In file `alu.`$i$`.`$j$`.`$k$`.nl`, i arithmetic operations, j comparison operations, and k logic operations are specified. The files in this set include

  `alu.2.0.0.nl alu.2.0.2.nl alu.2.0.8.nl alu.2.0.16.nl`
  `alu.2.4.0.nl alu.2.4.2.nl alu.2.4.8.nl alu.2.4.16.nl`
  `alu.2.8.0.nl alu.2.8.2.nl alu.2.8.8.nl alu.2.8.16.nl`
  `alu.4.0.0.nl alu.4.0.2.nl alu.4.0.8.nl alu.4.0.16.nl`
  `alu.4.4.0.nl alu.4.4.2.nl alu.4.4.8.nl alu.4.4.16.nl`
  `alu.4.8.0.nl alu.4.8.2.nl alu.4.8.8.nl alu.4.8.16.nl`

Each GENUS data file contains a number of similar VHDL specifications for GENUS components, such as that seen below taken from `alu.4.4.8.nl`.

```
Component ALU_16_4_4_8
  port (I0:    in  BIT_VECTOR (15 downto 0);
        I1:    in  BIT_VECTOR (15 downto 0);
        ICIN:  in  BIT;
        ISEL:  in  BIT_VECTOR (15 downto 0);
        O0:    out BIT_VECTOR (15 downto 0);
        OCOUT: out BIT;
        OREL:  out BIT);
end component;
attribute OPERATIONS of ALU_16_4_4_8: component
    is (ADD, SUB, INC, DEC, EQ, LT, GT,
        ZEROP, AND, OR, NAND, NOR, XOR,
        XNOR, LNOT, LIMPL);
```

The component name has the general form

type[<*no of inputs*>]_<*width*> [_<*no of operations*>]*

Thus, in the above specification `ALU` is the component type and `16` is its bit width; the component computes 4 arithmetic operations, 4 comparison operations, and 8 logic operations.

The <*no of inputs*> portion of the name is for bitwise components. For example, a specification from the file `mux4.nl` would look like

```
component MUX4_8
  port (I0:    in  BIT_VECTOR (7 downto 0);
        I1:    in  BIT_VECTOR (7 downto 0);
        I2:    in  BIT_VECTOR (7 downto 0);
        I3:    in  BIT_VECTOR (7 downto 0);
        ISEL:  in  BIT_VECTOR (3 downto 0);
        O0:    out BIT_VECTOR (7 downto 0));
end component;
```

The name `MUX4_8` indicates that this is a multiplexer with four 8-bit inputs.

Bibliography

Bergamaschi, R., "Automatic Synthesis and Technology Mapping of Combinational Logic," in *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD-88)*, November 1988, pp. 466–469.

Birmingham, W. P., A. P. Gupta, and D. P. Siewiorek, "The MICON System for Computer Design," in *Proceedings of the 26th Design Automation Conference*, June 1989, pp. 135–140.

Birmingham, W. P., and D. P. Siewiorek, "MICON: A Knowledge-Based Single Board Computer Synthesis Tool," in *Proceedings of the 21st Design Automation Conference*, June 1984.

Birmingham, W. P., and D. P. Siewiorek, *Automated Knowledge Acquisition for a Computer Hardware Synthesis System*, EDRC 18-06-88, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1988.

Birmingham, W. P., and D. P. Siewiorek, "Capturing Designer Expertise: The CGEN System," in *Proceedings of the 26th Design Automation Conference*, June 1989, pp. 610–613.

Boose, J. H., "Personal Construction Theory and The Transfer of Human Expertise," in *Proceedings of the National Conference on Artificial Intelligence (AAAI-84)*, August 1984, pp. 27–33.

Brayton, R. K., R. Camposano, G. De Micheli, R. H. J. M. Otten, and J. van Eijndhoven, "The Yorktown Silicon Compiler System," in D. D. Gajski (ed.), *Silicon Compilers*, Addison-Wesley, Reading, MA, 1988, pp. 204–310.

Brayton, R. K., and T. M. E. Detjens, S. Krishna, "Multiple-Level Logic Optimization System," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD-86)*, November 1986, pp. 356–359.

Brayton, R. K., G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vinventelli, *ESPRESSO-IIC: Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Netherlands, 1984.

Brayton, R. K., G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proceedings of the IEEE*, Vol. 78, No. 2, February 1990, pp. 264–300.

Brewer, F., and D. D. Gajski, "An Expert-System Paradigm for Design," in *Proceedings of the 23rd Design Automation Conference*, June 1986.

Brown, D. C., and B. Chandrasekaran, "Knowledge and Control for a Mechanical Design Expert System," *IEEE Computer*, Vol. 19, No. 7, July 1986, pp. 92–100.

Camposano, R., and L. H. Trevillyan, "The Integration of Logic Synthesis and High-Level Synthesis," in *Proceedings of the International Symposium of Circuits and Systems*, 1989, pp. 744–747.

Chen, G. D., and D. D. Gajski, "An Intelligent Component Database for Behavioral Synthesis," in *Proceedings of the 27th Design Automation Conference*, June 1990, pp. 150–155.

Cheng, E. K., and S. Mazor, "The Genesil Silicon Compiler," in D. D. Gajski (ed.), *Silicon Compilers*, Addison-Wesley, Reading, MA, 1988, pp. 361–405.

Davis, R., "Interactive Transfer of Expertise: Acquisition of New Inference Rules," *Readings in Artificial Intelligence*, 1981.

Davis, R., "Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases," in R. Davis, and D. B. Lenat (eds.), *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill, New York, NY, 1982.

de Geus, A. J., "Logic Synthesis Speeds ASIC Design," *IEEE Spectrum*, Vol. 26, No. 8, August 1989, pp. 27–31.

de Geus, A. J., and D. J. Gregory, "The Socrates Logic Synthesis and Optimization System," in G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti (eds.), *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, Martinus Nijhoff Publishers, Boston, MA, 1986, pp. 473–498.

de Man, H., J. Rabaey, J. Huisken, and J. Van Meerbergen, "CATHEDERAL-II: A Silicon Compiler of Digital Signal Processing," *IEEE Design and Test Magazine*, December 1986, pp. 13–25.

Detjens, E., G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology Mapping in MIS," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD-87)*, November 1987, pp. 116–119.

Director, S. W., A. C. Parker, D. P. Siewiorek, and D. E. Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems," *IEEE Transactions on Circuits and Systems*, Vol. CAS-28, No. 7, July 1981, pp. 634–645.

Dutt, N., *GENUS: A Generic Component Library for High-Level Synthesis*, technical report 88-22, Department of Information and Computer Science, University of California, Irvine, September 1988.

Friedman, T. D., and S. C. Yang, "Methods Used in an Automatic Logic Design Generator (ALERT)," *IEEE Transactions on Computers*, Vol. C-18, 1969, pp. 593–614.

Gajski, D. D., and F. D. Brewer, "Towards Intelligent Silicon Compilation," in G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti (eds.), *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, Martinus Nijhoff Publishers, Boston, MA, 1986, pp. 365–383.

Gajski, D. D., and D. E. Thomas, "Introduction to Silicon Compilation," in D. D. Gajski (ed.), *Silicon Compilers*, Addison-Wesley, Reading, MA, 1988, pp. 1–48.

Huhns, M. N., and R. D. Acosta, "Argo: A System for Design by Analogy," *IEEE Expert*, Fall 1988, pp. 53–68.

Johannsen, D., "Bristle Blocks: A Silicon Compiler," in *Proceedings of the 17th Design Automation Conference*, June 1979, pp. 310–313.

Kahn, G., S. Nowlan, and J. McDermott, "MORE: An Intelligent Knowledge Acquisition Tool," in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, August 1985, pp. 581–584.

Kelly, V. E., and L. I. Steinburg, "The Critter System: Analyzing Digital Circuits by Propagation Behaviors and Specifications," in *Proceedings of the National Conference on Artificial Intelligence (AAAI-82)*, July 1987, pp. 488–493.

Keutzer, K., "DAGON: Technology Binding and Local Optimization by DAG Matching," in *Proceedings of the 24th Design Automation Conference*, 1987, pp. 341–347.

Kipps, J. R., *The DTAS Component Generation System: Reference Manual and User's Guide*, technical report 91-77, Department of Information and Computer Science, University of California, Irvine, 1991.

Kipps, J. R., and D. D. Gajski, "The Role of Learning in Logic Synthesis," *International Journal of Pattern Recognition and Artificial Intelligence*, Vol. 4, No. 2, June 1990, pp. 167–180.

Klinker, G., C. Boyd, S. Genetet, and J. McDermott, "A KNACK for Knowledge Acquisition," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, July 1987, pp. 488–493.

Kowalski, T. J., *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, Boston, MA, 1985.

Lathrop, R. H., and R. S. Kirk, "A System Which Uses Examples to Learn VLSI Structure Manipulation," in *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, August 1986.

Lisanke, R., *Logic Synthesis and Optimization Benchmarks*, Microelectronics Center of North Carolina, Research Triangle Park, NC, 1988.

LSI, *CMOS Macrocell Manual*, LSI Logic, Inc., Milipitas, CA, 1987.

Mano, M. M., *Digital Logic and Computer Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

Marcus, S., J. McDermott, and T. Wang, "Knowledge Acquisition for Constructive Systems," in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, August 1985, pp. 637–639.

Marwedel, P., "The MIMOLA Design System: Tools for the Design of Digital Processors," in *Proceedings of the 21st Design Automation Conference*, June 1984, pp. 587–593.

McDermott, J., "R1: A Rule-Based Configurer for Computer Systems," *Artificial Intelligence*, Vol. 19, No. 1, September 1982.

McFarland, M. C., A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proceedings of the IEEE*, Vol. 78, No. 2, February 1990, pp. 301–318.

Minton, S., *Learning Effective Search Control Knowledge: An Explanation-Based Approach*, Ph.D. dissertation, Computer Science Department, Carnegie Mellon University, Pittsburg, PA, 1988.

Mitchell, T. M., S. Mahadevan, and L. I. Steinberg, "LEAP: A Learning Apprentice for VLSI Design," in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, August 1985, pp. 573–580.

Mitchell, T. M., L. I. Steinberg, and J. S. Shulman, "A Knowledge-Based Approach to Design," in *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, December 1984, pp. 27–34.

Mitchell, T. M., P. E. Utgoff, and R. B. Banerji, "Learning by Experimentation: Acquiring and Refining Problem-Solving Hueristics," in R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (eds.), *Machine Learning*, Tioga Publishing Company, Palo Alto, CA, 1983.

Mittal, S., C. L. Dym, and M. Morjaria, "PRIDE: An Expert System for the Design of Paper Handling Systems," *IEEE Computer*, Vol. 19, No. 7, July 1986, pp. 102–114.

Mittal, S., and F. Frayman, "Towards a Generic Model of Configuration Tasks," in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, August 1989, pp. 1395–1401.

Mostow, J., "Toward Better Models of the Design Process," *The AI Magazine*, Vol. 6, No. 1, Spring 1985, pp. 44–57.

Park, N., and A. C. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," *IEEE Transactions on Computer-Aided Design*, Vol. 7, No. 3, March 1989, pp. 356–370.

Shortliffe, E. H., *Computer-Based Medical Consultations: MYCIN*, Elsevier Publishers, New York, NY, 1976.

Soloway, E., J. Bachant, and K. Jensen, "Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule-Base," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, July 1987, pp. 824–829.

Southard, J. R., "MacPitts: An Approach to Silicon Compilation," *COMPUTER*, Vol. 16, No. 12, 1983, pp. 74–82.

Stefik, M., "Planning with Constraints (MOLGEN: Part 1)," *Artificial Intelligence*, Vol. 16, No. 2, May 1981.

Steinberg, L., "Design as Refinement Plus Constraint Propagation: The VEXED Experience," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, July 1987, pp. 830–835.

Thomas, D. E., "Automated Data Path Synthesis," in S. Goto (ed.), *Design Methodologies*, Elsevier Publishers, New York, NY, 1986.

Tong, C., and P. Franklin, "Tuning A Knowledge Base of Refinement Rules to Create Good Circuit Design," in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, August 1989.

van de Brug, A., J. Bachant, and J. McDermott, "The Taming of R1," *IEEE Expert*, Vol. 1, No. 3, 1986.

Vander Zanden, N., and D. D. Gajski, "MILO: Microarchitecture and Logic Optimizer," in *Proceedings of the 25th Design Automation Conference*, 1988, pp. 403–408.

Wolf, W. H., T. J. Kowalski, and M. C. McFarland, "Knowledge Engineering Issues in VLSI Design," in *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, August 1986, pp. 866–871.